LUNEX — VERSION 0.1 — DRAFT

LUNEX

A unified, object-oriented reference model for operational technology, safety, security, and AI.

lunex.cloud

Sixteen sub-models · Full specification, with diagrams

LUNEX Specification

Version 0.1 — Draft A unified, object-oriented reference model for operational technology, safety, security, and AI

Status of this document

This is a **draft specification**, authored by a single contributor and not yet reviewed by an independent working group. It is published to invite scrutiny, not to claim authority. Every design decision in this document is falsifiable — if you find a case it doesn't handle, that is more valuable feedback than agreement. See the repository's `README.md` for how to contribute.

Table of Contents

1. Introduction

LUNEX — *Layered Unified Namespace for Entities & eXtensibility*. Every object in the model, physical or not, is an **Entity** (`LunexObject`, Sub-model 1); the **Namespace** is the addressable path every one of them resolves to (Sub-model 2); **Layered** describes the four-layer structure the sixteen sub-models are organized into (§3); **eXtensibility** is why a seventeenth sub-model could be added without disturbing the sixteen already here.

1.1 Why LUNEX exists

Operational technology has no single reference model that spans its full breadth. PackML describes machine state behavior. ISA-88 describes batch process structure. ISA-95 describes the enterprise-to-control hierarchy. IEC 61508/61511 describe functional safety. IEC 62443 describes industrial cybersecurity. ISA-18.2 describes alarm management. Each is authoritative within its own scope, and each was written before the others, largely independent of them. An engineer building a modern OT system today must mentally stitch these together — and none of them anticipated the AI/data layer that is now a first-class part of the system, not an add-on.

LUNEX is an attempt to provide that missing unification: **one object model**, from a single Sensor up to the AI system that predicts, recommends, and — with approval — acts on the process, with every layer built on the same base class and the same design discipline.

1.2 Intentions

- **Completeness over minimalism.** LUNEX intentionally covers more ground than PackML or ISA-88 alone — process control, safety, security, alarm management, and AI/data are treated as one connected model, not five separate concerns bolted together.
- **Reuse before invention.** Every sub-model was tested against one question before introducing new vocabulary or structure: *does an existing standard already define this correctly?* Where the answer was yes (SIL, SIF, Security Levels, ISA-18.2's alarm states, IEC 60204-1's stop categories), LUNEX reuses it unchanged. New vocabulary was introduced only where no adequate term existed, or where an existing term already carried a conflicting meaning within this model.
- **One base class, no exceptions.** Every object in LUNEX — a pressure sensor, a safety interlock, a security zone, an alarm, an AI-generated recommendation — is a `LunexObject`. This is not a simplification for its own sake; it means every later capability (addressing, telemetry, rollup status, situational awareness) applies uniformly to everything in the model, including capabilities that didn't exist when the object class was first defined.
- **AI as a structural citizen, not an add-on.** Digital twins, predictive maintenance, scenario simulation, and closed-loop AI control are modeled with the same rigor and the same base class as a pressure transmitter — including explicit governance (`operatingBounds`, `requiresApproval`, `manualOverrideAvailable`) that treats autonomous behavior as something to be bounded and audited, not merely enabled.
- **A living document.** LUNEX is versioned and expected to change. Sub-model 8 exists specifically to keep every naming and design decision traceable, so that future revisions can be judged against the same discipline this version was built with.

1.3 What LUNEX is not

LUNEX does not replace IEC 61508/61511, IEC 62443, ISA-18.2, or IEC 60204-1 — it *reuses* them. An organization implementing LUNEX still needs a SIL/SIF assessment performed to IEC 61508/61511, still needs an IEC 62443 zone-and-conduit segmentation exercise, and so on. LUNEX's contribution is the object model that lets the *outputs* of those exercises live inside one consistent, addressable, machine-readable structure — not a replacement for the domain expertise those standards require.

2. Relationship to Existing Standards

Standard	What it defines	How LUNEX relates to it
ISA-88 (S88)	Batch process control structure (Process Cell, Unit, Equipment Module)	LUNEX's `Cell` (Sub-model 2) reuses S88's "Process Cell" concept directly, and `System` is the equivalent of S88's "Unit" — a functional grouping within a Cell, not a control system (DCS/SCADA/PLC). LUNEX's Assembly/Device/Component layers are a generalization of S88's structural pattern to non-batch processes.
ISA-95 (S95)	Enterprise-to-control hierarchy (Enterprise, Site, Area, Work Center)	LUNEX's `Area` (Sub-model 2) is S95's term, reused unchanged. `Realm` / `Domain` / `Location` replace S95's `Enterprise` / `Division` / `Site` — broadened beyond a single industrial plant, and chosen to enable the addressable namespace in Sub-model 7 (see Sub-model 8 for the full reasoning).
PackML (ISA-TR88.00.02)	Machine state behavior for packaging equipment	LUNEX's Sub-model 4 state machine is a generalization of PackML's state pattern to any object class, not only packaging machines.
IEC 61508 / IEC 61511	Functional safety, Safety Integrity Level (SIL), Safety Instrumented Function (SIF)	Reused **unchanged**. Sub-model 5 makes `Interlock` a first-class object specifically so it can carry the proof-test history IEC 61508/61511 require. A SIF is modeled as its own physically independent Assembly, per IEC 61511's independence requirement.
IEC 62443	Industrial cybersecurity: zones, conduits, Security Levels (SL)	Reused **unchanged**. Sub-model 6 introduces `Zone` and `Conduit` as first-class objects carrying IEC 62443's SL-Target/SL-Achieved directly.
ISA-18.2	Alarm management: priority, states, shelving, performance metrics	Reused **unchanged** for the concepts (priority, shelving, Return-to-Normal, performance metrics); Sub-model 10 defines the concrete `Alarm` object and its own state machine (deliberately not reusing Sub-model 4's, since operator-response states and object-behavior states answer different questions). Sub-model 14 computes the alarm performance metrics ISA-18.2 requires (alarms/hour, bad actors, time-to-acknowledge).
IEC 60204-1	Electrical safety of machinery, stop categories 0/1/2	Reused **unchanged**. Sub-model 5 confirms LUNEX's Emergency/Quick/Controlled Stopping states map directly to Category 0/1/2.
Purdue Enterprise Reference Architecture (PERA)	Layered model of enterprise/control network levels (L0–L4)	Sub-model 2's hierarchy is mapped onto Purdue levels at every rung, and Sub-model 6's security Zones commonly (not necessarily) align with a Purdue band.
OPC UA	Industrial interoperability protocol and information modeling	Not a structural dependency of LUNEX, but a natural transport/encoding candidate for the Sub-model 7 Telemetry Envelope in a real implementation. LUNEX does not mandate OPC UA; it deliberately stays protocol-agnostic.

3. How to Read This Specification

Each sub-model chapter (Sections 5–20) follows the same structure:

- **Purpose** — the functional problem this sub-model solves, in plain language, and why it couldn't be solved by an earlier sub-model alone.
- **Technical Description** — the formal object definitions: classes, attributes, state machines, and relationships to other sub-models.
- **Application Guidance** — how an implementer actually uses this sub-model: what to configure, what to compute, and a worked example.

Sub-models build on each other in the order presented. Sub-models 1–4 are foundational and assumed by everything after them. Sub-models 5–7 (safety, security, data/AI) are independent of each other but all depend on 1–4. Sub-models 9–13 depend on 7, 9, and 10 in combination. Sub-models 14–16 depend on 7 and, for 15–16, on 9–12.

The corresponding SVG diagram for each sub-model is referenced by filename; this document is the narrative and technical companion to those diagrams, not a replacement for them.

---

4. Glossary

Terms are listed alphabetically. Terms reused unchanged from an external standard are marked with that standard in brackets.

actionable — A derived boolean on `Alarm` (Sub-model 10): whether the operator can still change the outcome. Automatically computed from the linked Interlock's state — `true` while the Interlock is in Standby (not yet tripped), `false` once it has tripped.

Actuator — One of the five universal device classes (Sub-model 1); converts a control signal into physical action (valve, motor, heater, pump).

Alarm — A first-class `LunexObject` (Sub-model 10) representing a condition requiring operator attention. Priority is computed as severity × actionability, not severity alone.

AlarmResponseProcedure (ARP) — A reusable template (Sub-model 11), tied to an alarm type rather than an individual alarm instance, providing probable cause, consequence, corrective action, escalation, and a reference document. Mandatory for Priority 1/2 alarms.

Alarm.origin — Attribute on `Alarm` (Sub-model 15): `real-time | predictive`. Named `origin` rather than `source` specifically to avoid colliding with `Alarm.source` (Sub-model 10), which means "which Device."

Area — A level in the Sub-model 2 hierarchy, reused directly from ISA-95. A process area within a Location.

Assembly — A level in the Sub-model 2 hierarchy: a topology pattern (see Sub-model 3) composed of Devices.

AIControlUnit — A `ControlUnit` subclass (Sub-model 16) representing an AI model that actively adjusts process setpoints. Occupies an ordinary Control Unit slot in an Assembly; no new topology required.

automaticMode — Field on `SimulationPolicy` (Sub-model 12) and `RetentionPolicy` (Sub-model 14): `always | risk-based | none`. Paired with the universal `manualOverrideAvailable`.

bubbling — The Sub-model 9 principle that a parent object computes its Rollup from its direct children's Rollups only, never by scanning the full tree beneath it.

Cell — A level in the Sub-model 2 hierarchy, reused from ISA-88's "Process Cell."

Class — Attribute on `LunexObject`: a reference to the object's defining template in the organization's Inventory (Sub-model 1 §5.3), not a free-text string. Two objects share a `class` when instantiated from the same Inventory entry.

Cloud & Analytics — A capability flag in Sub-model 3: whether an Assembly's Control Unit(s) stream telemetry to the AI/Analytics layer (Sub-model 7). Independent of wiring shape.

Component — Abstract class (Sub-model 1) for parts of a Device that are not independently addressable (e.g., a digital input card).

Conduit — A first-class `LunexObject` (Sub-model 6), reusing IEC 62443: the governed communication pathway between two Zones, where a Firewall/IDS Control Unit typically sits.

Context Layer — Semantic relationships (`measures`, `partOf`, `semanticTag`) between objects (Sub-model 7), separate from and richer than the containment chain in Sub-model 2.

contributors — Field on a `Rollup` (Sub-model 9): the list of specific descendant objects responsible for a non-nominal severity tier.

Control Unit — One of the five universal device classes (Sub-model 1); performs logic/decision-making (PLC, DCS, microcontroller, and by extension `AIControlUnit`).

Device — Abstract class (Sub-model 1) for any independently addressable, assembly-level unit; composes zero or more Components. Optionally carries `physicalRef` when its physical hardware is shared with other functionally distinct Devices.

Digital Twin — The AI-side, queryable mirror of a physical object (Sub-model 7), kept in sync via the Telemetry Envelope. Holds only the *current* state — see Historian for historical state.

Domain — A level in the Sub-model 2 hierarchy: an optional business-unit grouping between Realm and Location.

Envelope — See Telemetry Envelope.

GuidanceRecommendation — A first-class `LunexObject` (Sub-model 12) holding a set of `ScenarioResult`s. `source` (later extended with a third value in Sub-model 15) indicates whether the guidance is `procedural`, `simulated`, or

`predictive`. `state: Open | Applied | Dismissed | Expired` — overrides `methods()` with `apply()` / `dismiss()`; expires automatically as the process state it was generated against moves on.

goldenScenarioId — Field on `GuidanceRecommendation`: the id of the recommended `ScenarioResult`, or `null` if the scenarios are equivalent and no single best option is being asserted.

Health — Type of `LunexObject.health` (Sub-model 1): `{ score : 0-100, flags : string[] }`. Diagnostic condition, deliberately independent of `state` — an object can be `On` and simultaneously report degraded health.

Historian — An append-only record (Sub-model 14) of every Telemetry Envelope ever emitted, alongside (not gating) the Digital Twin.

HistorianRecord — One row in the Historian: path, timestamp, state, health, properties, and a reference to the `RetentionPolicy` governing its resolution.

High-Availability (HA) — A capability flag in Sub-model 3: whether an Assembly's Control Unit(s) are deployed as an active/standby redundant pair. Independent of wiring shape.

id — Attribute on `LunexObject` (Sub-model 1): the system-generated, permanent identifier. Never reused, never renamed. Every `Ref` field in this specification (`DeviceRef`, `AlarmRef`, `ZoneRef`, etc.) points to an `id`, not a `tag`.

Inhibited — A Sub-model 4 state: Tier 1 (Restricted) per Sub-model 9. Reached from Standby (via Inhibiting) or from On (via Emergency Stopping). Requires an explicit Clear to leave.

ImprovementRecommendation — A first-class `LunexObject` (Sub-model 15) suggesting a configuration change based on a Historian pattern. `requiresApproval` is always `true` — no confidence threshold ever skips human review.

Interlock — A first-class `LunexObject` (Sub-model 5), reused conceptually from IEC 61508/61511: a safety condition that forces or blocks a transition on a target Device. Carries its own proof-test history.

Inventory — The Sub-model 1 term for a reusable class/template (chosen over "Library" for plainer, more approachable naming).

jumpToWorst() — A navigation function (Sub-model 13): from any hierarchy node, follows the Rollup's own `contributors` path directly to the Device/Alarm causing the worst status, bypassing manual layer-by-layer navigation.

Locked — A Sub-model 4 state: Tier 1 (Restricted). Reached from Off (via Locking), or from Standby/Inhibited (via Disabling → Off → Locking). The target of `lockType: lock`, and of `lockType: inhibit` when the object is already Off (since Inhibited is unreachable from Off).

lockType — Attribute on `Interlock` (Sub-model 5): `inhibit | lock`. Determines which Sub-model 4 transition the Interlock forces, given the object's current state.

Location — A level in the Sub-model 2 hierarchy: a plant, building, vessel, or remote site.

LunexObject — The abstract base class (Sub-model 1) every object in LUNEX inherits from, without exception.

manualOverrideAvailable — Field on `SimulationPolicy` and `RetentionPolicy`: always `true`, regardless of `automaticMode`. Manual triggering is universal, not exclusive to a specific mode.

methods() — Attribute on `LunexObject` (Sub-model 1 §5.2.1): a computed view, not a stored list — returns the Sub-model 4 transition-triggers valid from the object's current `state`, excluding `Complete`. Empty for any transient state. `Alarm`, `Interlock`, `GuidanceRecommendation`, `PredictedEvent`, and `ImprovementRecommendation` override it with their own trigger sets.

operatingBounds — Attribute on `AIControlUnit` (Sub-model 16): required whenever `target` is the physical Device; may be empty only when `target` is the Digital Twin, since an unbounded action there cannot cause real-world harm.

Operator — A first-class `LunexObject` peer (Sub-model 1): `role : string`. Deliberately minimal — records identity and capacity, not permissions; resolves `OperatorRef` wherever it's used elsewhere in the model.

parent — Attribute on `LunexObject` (Sub-model 1): the immediate container in the Sub-model 2 containment chain. Distinct from `physicalRef` (shared hardware) and from any Sub-model 4 transition relationship — three separate kinds of "points to another object."

PredictedEvent — A first-class `LunexObject` (Sub-model 15) representing a forward-looking claim derived from a Historian pattern. Raises into the ordinary Alarm system, tagged `PREDICTIVE`, rather than occupying a separate screen.

physicalRef — Optional attribute on `Device` (Sub-model 1): `DeviceRef | ComponentRef`, points to the actual shared, serviceable unit — a Device for simple hardware, or a specific Component (e.g. one CPU in a multi-CPU rack) when the Device's internal structure means different functional roles are backed by different sub-parts. Also confirms Sub-model 3's Integrated topology when shared across an entire chain.

priority — Attribute on `Alarm` (Sub-model 10): 1 (Critical) to 4 (Log Only), computed as severity × actionability.

properties — Attribute on `LunexObject` (Sub-model 1): `Map<Key, Value>` — the object's own data, including the current output of any continuous operational functionality (Sub-model 1 §5.2.2). Specific to the object's `class`; not standardized in shape beyond the map itself.

Realm — The root level of the Sub-model 2 hierarchy: the whole organization's namespace.

requiresApproval — Attribute on `ImprovementRecommendation`: fixed to `true`. Has no false case, even at maximum confidence.

resetMode — Attribute on `Alarm` (Sub-model 10): `auto | manual`. Determines whether an alarm returns to Normal automatically once its condition clears, or requires an explicit operator reset via the Cleared (RTN) state.

riskThreshold — Field on `SimulationPolicy` / `RetentionPolicy` used when `automaticMode: risk-based`: a Sub-model 9 Tier value above which the automatic behavior triggers.

Rollup — A computed view (Sub-model 9), not a stored class: `worstTier`, `tierCounts`, and `contributors` summarizing a node's own state plus its children's Rollups.

ScenarioResult — A single simulated outcome (Sub-model 12) within a `GuidanceRecommendation`: description, predicted outcome, success probability, and whether it is the golden scenario.

Sensor — One of the five universal device classes (Sub-model 1); converts a physical quantity into a raw signal.

severity tier — See Tier.

shelvedUntil — Attribute on `Alarm`: the timestamp a Shelved alarm automatically returns to Unacknowledged. Shelving always expires; it cannot permanently silence an alarm.

Signal Converter — One of the five universal device classes (Sub-model 1); converts a control signal into a physical/power signal for the Actuator side.

SIF (Safety Instrumented Function) [IEC 61511] — Reused unchanged. Modeled in Sub-model 5 as an Assembly with `independent: true`, physically separate from the BPCS Assembly.

SIL (Safety Integrity Level) [IEC 61508] — Reused unchanged as a property of a SIF Assembly.

SimulationPolicy — Governs when normal operation (not just alarms) triggers scenario simulation (Sub-model 12). See `automaticMode`, `riskThreshold`, `manualOverrideAvailable`.

SL (Security Level) [IEC 62443] — Reused unchanged, as `securityLevelTarget` / `securityLevelAchieved` on a `Zone` or `Conduit`.

Standby — A Sub-model 4 state: Tier 3 (Nominal) per Sub-model 9. Never a target of an Interlock.

State — The 15-value enumeration defined in Sub-model 4 (5 stable, 10 transient), shared by every `LunexObject` unless the object's class doesn't need a given branch.

System — A level in the Sub-model 2 hierarchy: a functional unit within a Cell, equivalent to ISA-88's Unit. Not a control system (DCS/SCADA/PLC) — that would be a `Control Unit` (Sub-model 1) instance living inside this level.

tag — Attribute on `LunexObject` (Sub-model 1): the human/engineering-facing label (e.g. `PT-101`). Unlike `id`, `tag` can be renamed during the object's life without breaking any reference, since references use `id`.

target — Attribute name reused across two classes with different meanings, the same way `state` is: on `Interlock` (Sub-model 5), `target : DeviceRef` is the Device the Interlock forces or blocks a transition on; on `AIControlUnit` (Sub-model 16), `target : PhysicalDevice | DigitalTwin` is what its output is routed to. No object is ever both classes at once, so the two meanings never collide in practice — documented here rather than resolved by renaming, consistent with how `Alarm.state`'s own value space coexists with `Interlock.state`'s reuse of the Sub-model 4 enum.

Telemetry Envelope — The addressed, timestamped wrapper (Sub-model 7) around what `LunexObject` already exposes (class, state, health, properties). Streams to both the Digital Twin and the Historian.

Tier — The severity ranking defined in Sub-model 9: 0 (Critical) to 3 (Nominal), ranking the Sub-model 4 states by severity — a ranking the state machine itself does not carry.

Transducer — One of the five universal device classes (Sub-model 1); converts a raw sensor signal into a usable

signal.

tierCounts — Field on a `Rollup`: a count of descendants at each Tier, preserving multi-condition situations a single worst-case value would hide.

votingArchitecture — Mandatory attribute on a SIF Assembly (Sub-model 5): `MooN` per layer (sensor, logicSolver, finalElement), reused unchanged from IEC 61508. Mandatory, unlike `physicalRef`, because an implicit default here would silently assert a SIL-relevant fact.

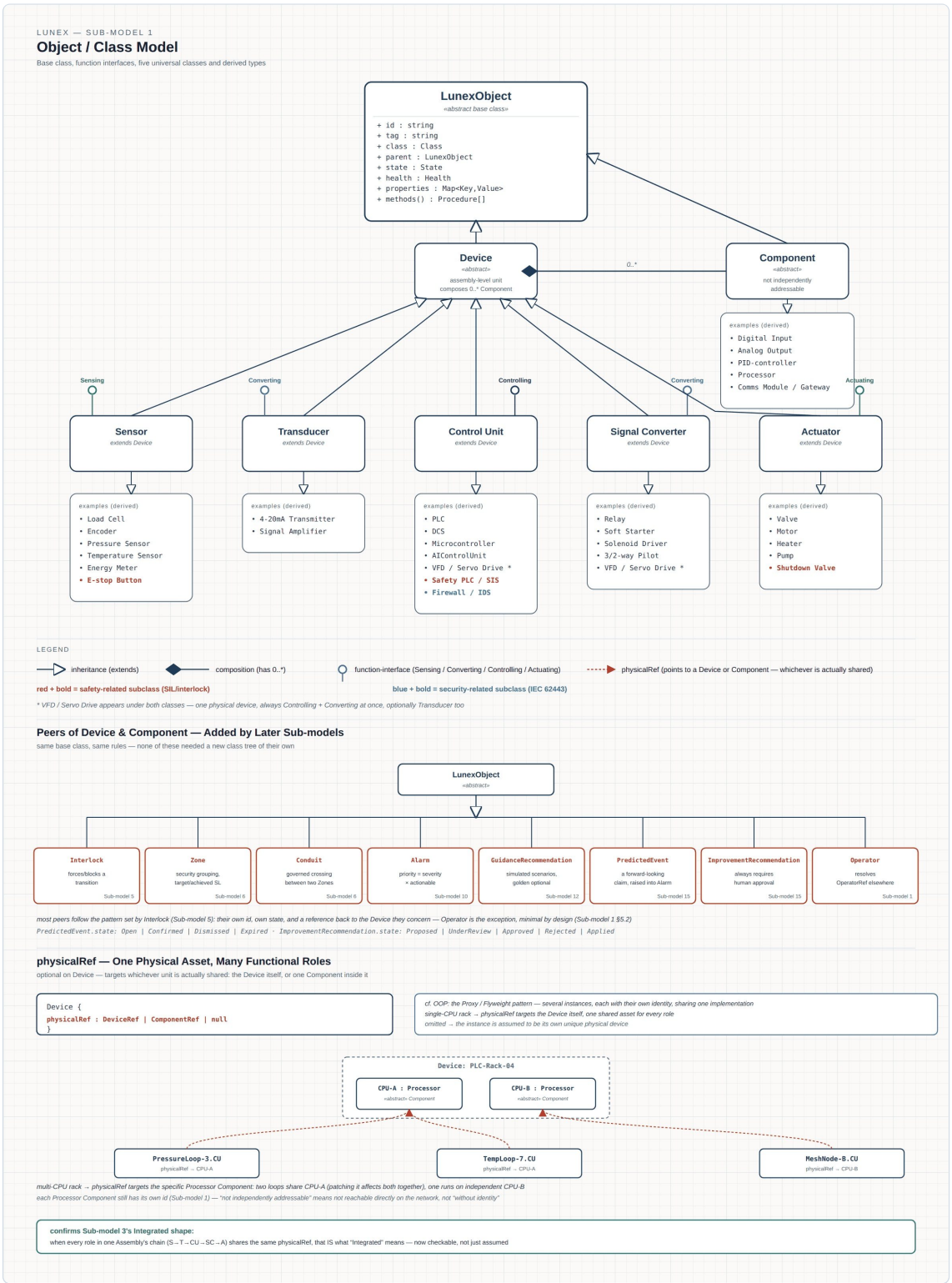
worstTier — Field on a `Rollup`: the lowest (worst) Tier present anywhere beneath a node.

Zone — A first-class `LunexObject` (Sub-model 6), reusing IEC 62443: a security grouping with a target and achieved Security Level. Not to be confused with the Sub-model 2 hierarchy, which uses `Area` instead specifically to keep this meaning unambiguous.

---

5. Sub-model 1 — Object / Class Model

Diagram: `lunex-object-model.svg`



lunex-object-model.svg

5.1 Purpose

Every OT reference model needs a base unit of "thing." PackML and ISA-88 each define their own, scoped to their own domain (a packaging machine state; a batch equipment module). LUNEX needs one that scales from a single pressure transmitter to an AI recommendation engine, without a different modeling approach at each scale. Sub-model 1 defines that base unit once, and everything built afterward — safety, security, alarms, AI — is a subclass or a peer of it, never a parallel system.

5.2 Technical Description

`LunexObject` (abstract base class). Every object in LUNEX, without exception, has:

```
LunexObject {
  id            : string
  tag           : string
  class         : Class
  parent        : LunexObject
  state         : State                 (Sub-model 4)
  health        : Health
  properties    : Map<Key, Value>
  methods()     : Procedure[]
}
```

- **`id`** — the system-generated, permanent identifier. Never reused, never renamed. This is what every `Ref` field elsewhere in this specification (`DeviceRef`, `AlarmRef`, `ZoneRef`, and so on) actually points to.
- **`tag`** — the human/engineering-facing label (e.g. `PT-101`, `PressureLoop-3.CU`). Unlike `id`, `tag` can be renamed during the object's life (a re-tagging project, a loop renumbering) without breaking any reference, because references use `id`.
- **`class`** — a reference to the object's defining template in the organization's Inventory (Sub-model 1 §5.3), not a free-text string. Two objects share a `class` when they were instantiated from the same Inventory entry.
- **`parent`** — the immediate container in the Sub-model 2 containment chain (e.g. a `Device`'s `parent` is its `Assembly`). This is structural containment, not `physicalRef` (Sub-model 1's optional shared-hardware pointer) and not the Sub-model 4 state machine's own transition graph — three separate relationships that happen to all involve one object pointing at another.
- **`health : Health`** — `{ score : 0-100, flags : string[] }`. Diagnostic condition, deliberately independent of `state`: an object can be `On` (Sub-model 4) and simultaneously report degraded `health` (e.g. a sensor drifting out of calibration) without that affecting which state-transitions are available.
- **`properties`** — the object's own data: current value, unit, setpoint, whatever is specific to its `class`. This is also where continuous operational output lives (see §5.2.1 below).
- **`methods()`** — see §5.2.1.

`state` and `health` are deliberately separate: `state` is the Sub-model 4 operating mode; `health` is diagnostic condition, independent of mode.

5.2.1 `methods()` — Derived from State, Not a Free List

`methods()` is not an arbitrary, per-class method table. It is a **computed view**: the set of Sub-model 4 transition-triggers valid from the object's *current* `state`, minus `Complete` (which is an automatic, system-internal transition — never something a caller invokes). Calling `methods()` on an object in `Standby` and the same object in `On` returns two different, non-overlapping lists, because the underlying state machine (Sub-model 4) only exposes different outgoing edges from each.

Canonical trigger names, taken directly from the Sub-model 4 transition labels already established:

From state	`methods()` returns
`Locked`	`unlock()`
`Off`	`lock()`, `enable()`
`Standby`	`start()`, `disable()`, `inhibit()`
`Inhibited`	`clear()`, `disable()`
`On`	`controlledStop()`, `quickStop()`, `emergencyStop()`
any transient state (`Starting`, `Enabling`, `Disabling`, `Unlocking`, `Locking`, `Inhibiting`, `Clearing`, all three Stopping variants)	*(empty)*

The empty case for transient states is not an oversight — it follows directly from Sub-model 4's own graph, which was already drawn this way: while an object is completing its own transition, it accepts no new command until that transition reaches a stable state. This specification simply makes the rule explicit rather than leaving it implicit in the diagram.

Classes with their own state machine override `methods()` with their own trigger set, rather than inheriting the Sub-model 4 list:

Class	Sub-model	Override triggers
`Alarm`	10	`acknowledge()`, `shelve()`, `operatorReset()`, `suppress()` / `unsuppress()`, `takeOutOfService()` / `returnToService()`
`Interlock`	5	`clear()` (releases a tripped condition once the underlying cause is gone; distinct from the target Device's own `clear()`)
`GuidanceRecommendation`	12	`apply()` (Open → Applied), `dismiss()` (Open → Dismissed). `Expired` happens automatically as the underlying process state moves on, never via a method call.
`PredictedEvent`	15	`confirm()` (Open → Confirmed), `dismiss()` (Open → Dismissed). `Expired` is never a method call — it happens automatically once `predictedWindow` passes with no confirmation.
`ImprovementRecommendation`	15	`startReview()` (Proposed → UnderReview), `approve()` / `reject()` (UnderReview → Approved / Rejected), `apply()` (Approved → Applied). No method ever skips `UnderReview` — see Sub-model 15 §19.2, `requiresApproval`.
`AIControlUnit`	16	inherits the Sub-model 4 list unchanged — `enable()` / `disable()` are exactly `enable()` / `disable()` from the table above, since an AIControlUnit's own On/Off/Standby is the ordinary state machine, not a new one (Sub-model 16 §20.2)

5.2.2 Scope Boundary: Continuous Operational Functionality

`methods()` covers discrete, externally-triggered state transitions. It deliberately does **not** cover continuous operational functionality — a PID loop recalculating every scan cycle, a Sensor sampling continuously, an AIControlUnit's own control algorithm running while it is `On`. This is not a gap; it is the same boundary this specification draws everywhere else: Sub-model 5 never specifies *how* a SIF computes its trip logic, only that it must; Sub-model 16 never specifies *which* algorithm an AIControlUnit runs, only its `objective` (intent, not implementation) and the governance around it. LUNEX is a structural and governance model, not a control-algorithm standard.

What continuous functionality needs is already covered, without any new mechanism:

Need	Already covered by
The current output of that continuous functionality	`properties` (this section)
Getting that output off the object	Telemetry Envelope (Sub-model 7)
Whether it's allowed to run at all	`state` + Interlock (Sub-model 5) + `operatingBounds` (Sub-model 16)
What it's trying to achieve, without prescribing how	`objective` (Sub-model 16, applicable to any Control Unit, not only `AIControlUnit`)

`Device` (abstract, extends `LunexObject`) — an independently addressable, assembly-level unit. Composes zero or more `Component`.

```
Device {
  physicalRef : DeviceRef | ComponentRef | null
}
```

`physicalRef` is optional — most Devices don't need it, and omitting it means the instance is assumed to be its own unique physical asset. It exists for the common case where one physical unit plays several distinct functional roles: a single PLC rack, for example, may be the Control Unit for three unrelated loops at once (`PressureLoop-3.CU`, `TempLoop-7.CU`, `MeshNode-B.CU` in a Mesh Assembly), each with its own `id`, `tag`, and `state`, all three pointing back to the same `physicalRef`. Conceptually this mirrors the Proxy/Flyweight pattern in OOP: several instances, each with their own identity and interface, sharing one underlying implementation they don't own. It also gives Sub-model 3's **Integrated** topology a checkable definition rather than an assumed one: when every role in one Assembly's chain (S→T→CU→SC→A) shares the same `physicalRef`, that *is* what Integrated means.

`physicalRef` targets whichever granularity is actually the serviceable unit — not always the whole Device. A PLC rack with a single CPU is one shared asset, so `physicalRef` there points to the rack `Device` itself. A rack with **two independent CPUs** is not one shared asset — patching CPU-A never affects loops running on CPU-B — so `physicalRef` should point to the specific `Processor` `Component` (already a standard Component example, Sub-model 1 §5.2) rather than the rack. If `PressureLoop-3.CU` and `TempLoop-7.CU` both run on CPU-A while `MeshNode-B.CU` runs on CPU-B, the first two share one `physicalRef` and the third has a different one, even though all three physically live in the same rack. This works without any new rule: `Component` already carries its own `id` and `tag` (inherited from `LunexObject`) — "not independently addressable" (Sub-model 1 §5.2) describes network reachability, not identity, so a `Component` is just as valid a `physicalRef` target as a `Device`.

`Component` (abstract, extends `LunexObject`) — a part of a Device that is *not* independently addressable (e.g., a digital input channel on a PLC card). Examples: Digital Input, Analog Output, PID-controller, Processor, Comms Module / Gateway.

Five universal device classes (all extend `Device`), distinguished by composable function-interfaces rather than fixed categories:

Class	Function-interface	Role
`Sensor`	Sensing	physical quantity → raw signal
`Transducer`	Converting	raw signal → usable signal
`Control Unit`	Controlling	logic / decision
`Signal Converter`	Converting	control signal → physical/power signal
`Actuator`	Actuating	signal → physical action

Function-interfaces are composable, not exclusive — a class may implement more than one. This is why an IoT-capable smart device is not a sixth class: it is an existing class (e.g. `Sensor`) plus a `Comms Module` `Component`, or a fully-Integrated topology (Sub-model 3).

Eight peers of `Device` / `Component`, added by later sub-models, each extending `LunexObject` directly (not `Device`):

Peer	Sub-model	Summary
`Interlock`	5	Forces/blocks a Sub-model 4 transition
`Zone`	6	Security grouping with target/achieved SL
`Conduit`	6	Governed crossing between two Zones
`Alarm`	10	Priority = severity × actionability
`GuidanceRecommendation`	12	A set of simulated scenarios
`PredictedEvent`	15	A forward-looking claim, raised into Alarm
`ImprovementRecommendation`	15	Always requires human approval
`Operator`	1	A human referenced by `OperatorRef` elsewhere in the model

`Operator` closes a gap the model had without formalizing: `OperatorRef` is used elsewhere (`AIControlUnit.disabledBy`, Sub-model 16) as if it resolved to something, but nothing ever defined what. It is deliberately minimal:

```
Operator extends LunexObject {
  role : string   (e.g. "Control Room Operator", "Safety Engineer", "Maintenance Engineer")
}
```

`role` is a plain descriptive string, not a permissions or access-control system — LUNEX records *who* took an action (via `id`, inherited from `LunexObject`) and *in what capacity* (`role`), not *what they were authorized to do*. A full role-based access model is a genuinely separate concern from object structure and is intentionally left for a future sub-model, not bolted on here to close this gap. Every other place in this specification that refers informally to "the operator" (acknowledging an alarm, approving an `ImprovementRecommendation`, applying a `GuidanceRecommendation`) is not retroactively required to hold a formal `OperatorRef` — that reference is only mandatory where a schema explicitly types a field as `OperatorRef`, as `AIControlUnit.disabledBy` already does.

Color/marking convention (used consistently across all sub-model diagrams): a derived class marked red-and-bold is safety-related (SIL/Interlock); marked blue-and-bold is security-related (IEC 62443).

5.3 Application Guidance

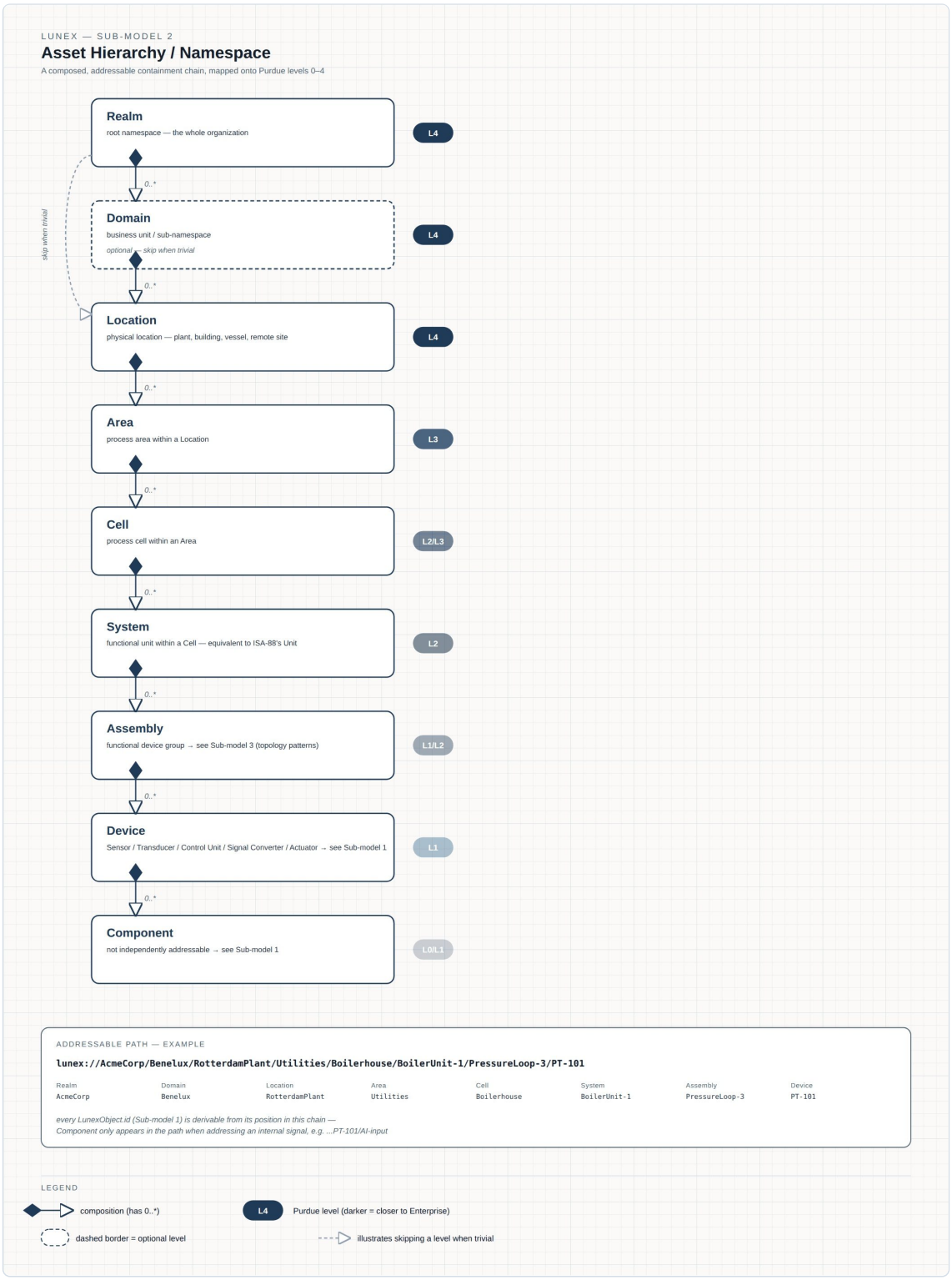
To model a new physical device:

1. Identify its function(s) — Sensing, Converting, Controlling, Actuating — and pick the matching universal class(es). Most real devices need only one; an intelligent transmitter that also converts is the exception, not the rule. A variable-frequency or servo drive is a canonical multi-interface case: it is always both a `Control Unit` (it computes the output from setpoint and feedback) and a `Signal Converter` (it produces the variable-frequency power signal that drives the motor), and a `Transducer` too if its processed feedback is exposed externally rather than staying internal to its own loop.
2. Decide whether it is independently addressable on the network (`Device`) or a sub-part of one (`Component`).
3. Define it as a derived class in your organization's Inventory (a reusable template — see Sub-model 8's reasoning for why "Inventory" rather than "Library"), not as a one-off instance.
4. Only introduce a new peer class (alongside Interlock, Zone, etc.) if the object genuinely needs its own identity, state, and audit trail independent of any Device — most new concerns should first be checked against whether they can be a `Component`, a `property`, or an `Assembly` capability flag instead.
5. Set `physicalRef` only when it's actually needed — when one physical asset genuinely backs more than one functional tag. Don't set it by default; an unnecessary `physicalRef` implies a sharing relationship that doesn't exist and will mislead maintenance planning rather than help it.

---

6. Sub-model 2 — Asset Hierarchy / Namespace

Diagram: `lunex-asset-hierarchy.svg`



lunex-asset-hierarchy.svg

6.1 Purpose

An object model is only useful if every instance can be found. ISA-95 provides a hierarchy, but one scoped to enterprise manufacturing; LUNEX needed a namespace broad enough to address anything from a multinational's plant network down to a single I/O channel, in a form suited to machine addressing (not just human reporting).

6.2 Technical Description

A **composed containment chain**, each level composing zero-or-more of the next:

```
Realm → Domain → Location → Area → Cell → System → Assembly → Device → Component
```

Level	Purdue	Role
Realm	L4	root namespace — whole organization
Domain	L4	business unit (optional — skip when trivial)
Location	L4	plant, building, vessel, remote site
Area	L3	process area within Location (ISA-95 term, reused)
Cell	L2/L3	process cell (ISA-88 term, reused)
System	L2	functional unit within a Cell (equivalent to ISA-88's Unit)
Assembly	L1/L2	topology pattern (Sub-model 3)
Device	L1	one of the five universal classes (Sub-model 1)
Component	L0/L1	not independently addressable

Any level may be skipped when trivial (Domain is the most commonly skipped). Non-production concerns (utilities, safety, IoT) use the same `Area` level as production — there is no separate branch.

Addressable path: every `LunexObject.id` is derivable from its position in this chain:

```
lunex://AcmeCorp/Benelux/RotterdamPlant/Utilities/Boilerhouse/BoilerUnit-1/PressureLoop-3/PT-101
```

`Component` appears in the path only when addressing an internal signal.

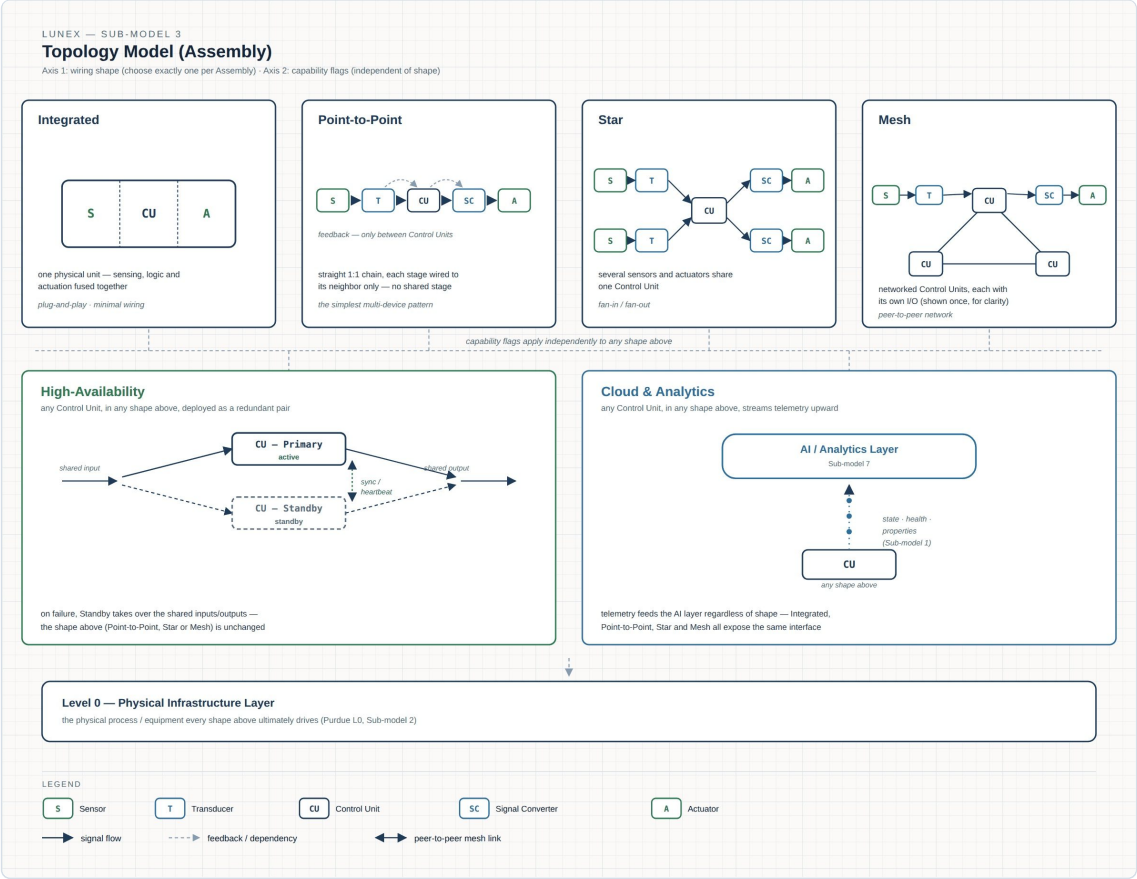
6.3 Application Guidance

1. Start from the top (Realm) and work down; skip Domain immediately if the organization has no meaningful business-unit split relevant to OT addressing.
2. Assign one `Area` per physically or organizationally distinct process zone — utilities, safety systems, and IoT devices each get their own Area rather than a special-cased branch.
3. Use the resulting path directly as the `path` field in the Sub-model 7 Telemetry Envelope — no separate addressing scheme is needed.
4. When integrating with an existing ISA-95 deployment, map `Area` and `Cell` directly (they are the same concept); map `Enterprise` / `Site` to `Realm` / `Location`, noting the broadened scope.

---

7. Sub-model 3 — Topology Model (Assembly)

Diagram: `lunex-topology-model.svg`



lunex-topology-model.svg

7.1 Purpose

PackML and most DCS documentation implicitly assume a single wiring pattern (roughly Point-to-Point). Real plants mix integrated smart devices, star-wired I/O, and networked mesh controllers within the same facility, often the same Cell. Sub-model 3 makes wiring shape an explicit, independent property of an Assembly rather than an unstated assumption.

7.2 Technical Description

Axis 1 — Wiring shape (exactly one per Assembly):

Shape	Pattern
Integrated	Sensor + Control Unit + Actuator as one physical unit
Point-to-Point	1:1 linear chain: Sensor → Transducer → Control Unit → Signal Converter → Actuator
Star	Multiple sensors/actuators share one Control Unit (fan-in/fan-out)
Mesh	Multiple Control Units networked peer-to-peer, each with its own I/O

All four shapes include the full S→T→CU→SC→A chain — Transducers and Signal Converters are present in Star and Mesh too, not just Point-to-Point.

Axis 2 — Capability flags (0, 1, or both; independent of shape):

- **High-Availability (HA)** — Control Unit(s) deployed as an active/standby redundant pair.
- **Cloud & Analytics** — Control Unit(s) stream telemetry to the AI/Analytics layer (Sub-model 7).

An `AIControlUnit` (Sub-model 16) occupies a Control Unit slot in any of these shapes — most commonly Point-to-Point or Star — without requiring a new shape.

Integrated, made checkable: "one physical unit" no longer has to be an assumption. When every role present in an Assembly's chain shares the same `Device.physicalRef` (Sub-model 1), that shared reference *is* the Integrated shape — a Mesh Assembly whose three nodes each happen to be their own separate hardware is not Integrated no matter how tightly they're networked; a Point-to-Point chain whose S/CU/A all resolve to one `physicalRef` is Integrated even if no one flagged it as such.

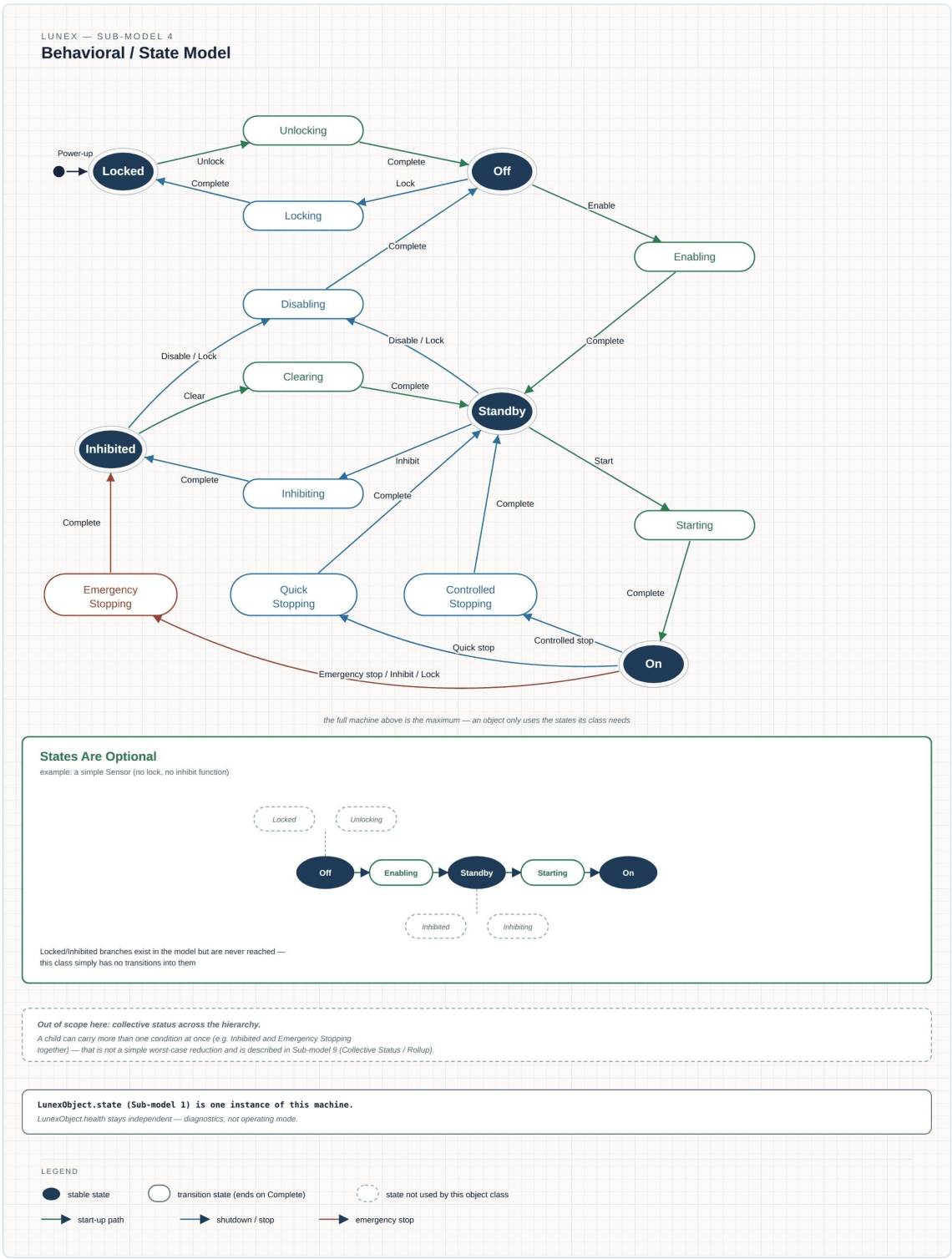
7.3 Application Guidance

1. For each Assembly, first pick the wiring shape by asking: does one Control Unit own this signal chain alone (Point-to-Point/Star), do multiple Control Units coordinate (Mesh), or is it one sealed unit (Integrated)?
2. Then, independently, decide HA and Cloud & Analytics per Assembly — these are operational/business decisions, not wiring decisions, and should not force a shape change.
3. An Assembly flagged `independent: true` (Sub-model 5, SIF) must never share a Control Unit with a non-independent Assembly, regardless of shape.

---

8. Sub-model 4 — Behavioral / State Model

Diagram: `lunex-state-model.svg`



lunex-state-model.svg

8.1 Purpose

PackML defines a state machine for packaging machines. LUNEX needs the equivalent for *any* object class — a Sensor, a Control Unit, an Interlock, an AIControlUnit — sharing the same vocabulary so that “what state is this in” always means the same thing, regardless of what kind of object is being asked.

8.2 Technical Description

A single state machine, shared by every `LunexObject`, backing `LunexObject.state`:

Stable states: Locked, Off, Standby, Inhibited, On **Transition states**: Unlocking, Locking, Enabling, Disabling, Starting, Inhibiting, Clearing, Quick Stopping, Controlled Stopping, Emergency Stopping

Key transition paths: - Start-up: Locked → Unlocking → Off → Enabling → Standby → Starting → On - Shutdown: On → (Controlled/Quick/Emergency Stopping) → Standby or Inhibited - Lock/Unlock: Locked ↔ (Unlocking/Locking) ↔ Off - Inhibit/Clear: Standby ↔ (Inhibiting/Clearing) ↔ Inhibited - Disable: Standby or Inhibited → Disabling → Off

Transition labels reflect every purpose that triggers them, since Sub-model 5's Interlock reuses these exact transitions rather than inventing new ones: - `Emergency stop / Inhibit / Lock` — the On → Emergency Stopping transition, used by a genuine emergency stop, an inhibit-type Interlock, and (as the first leg) a lock-type Interlock, all starting from On. - `Disable / Lock` — the Standby/Inhibited → Disabling transition, used by a genuine disable action and by a lock-type Interlock's path to Off → Locked.

Two design decisions made explicit: 1. **States are optional per object capability.** A simple Sensor only ever uses Off → Enabling → Standby → Starting → On; the Locked/Inhibited branches exist in the model but are simply never reached for that class. 2. **State is always independently owned.** A parent's own state is never overridden by a child's state. Collective status across a hierarchy is a separate, deliberately deferred concern — resolved in Sub-model 9, not here.

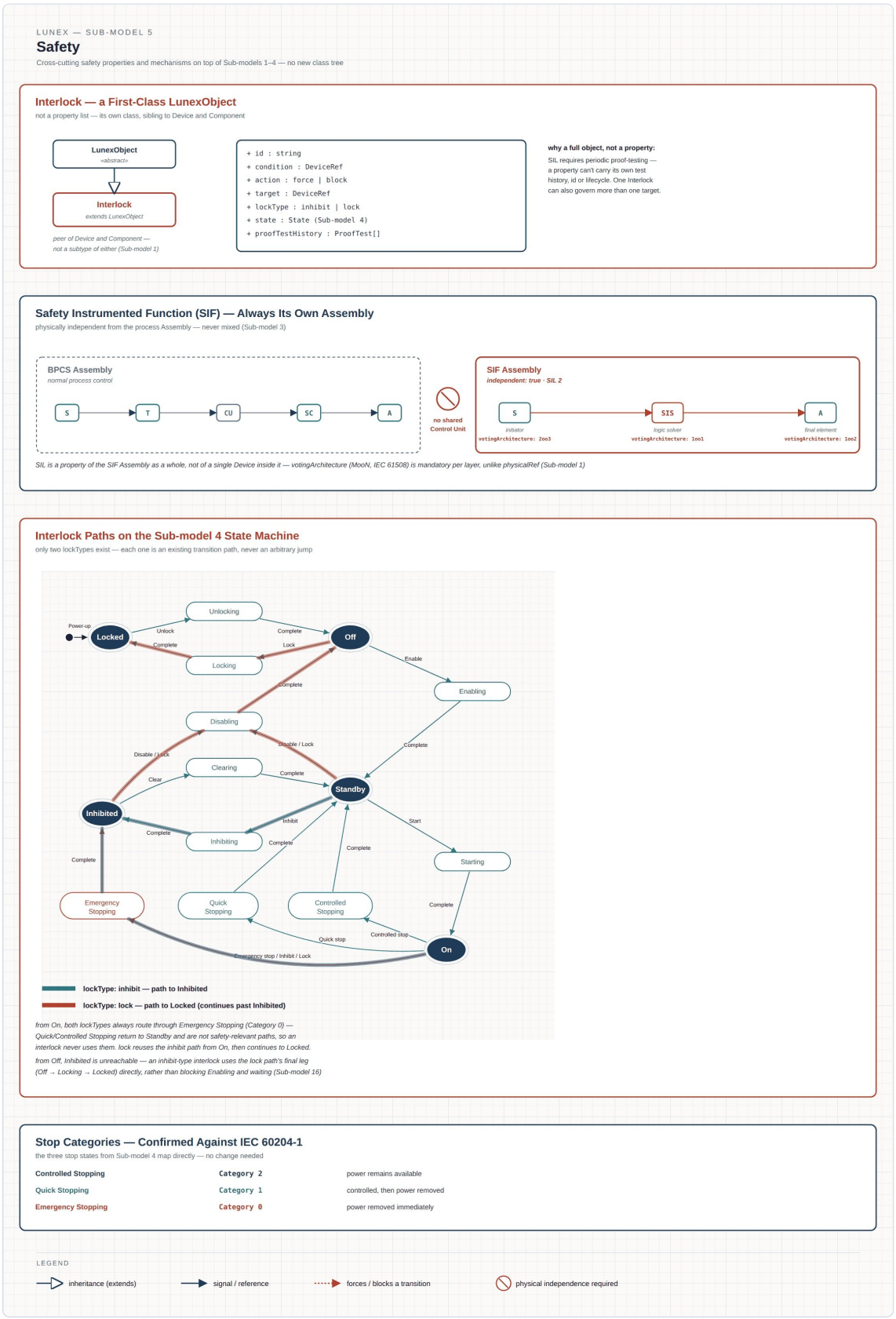
8.3 Application Guidance

1. When defining a new derived class, identify only the states it actually needs — do not force every class through the full 14-state machine if, e.g., it has no safety-lockout requirement.
2. Do not build a parallel or simplified state machine for a new object type (as was mistakenly done in an early draft of Sub-model 16) — transplant this state machine and highlight only the relevant paths. This guarantees any correction made here propagates everywhere it's reused.
3. Do not attempt to compute a hierarchy-wide "worst state" from this sub-model alone; see Sub-model 9.

---

9. Sub-model 5 — Safety

Diagram: `lunex-safety-model.svg`



lunex-safety-model.svg

9.1 Purpose

Functional safety (IEC 61508/61511) needs traceable, auditable interlock logic — proof-test history, independence from the BPCS, and precise, unambiguous behavior. Modeling a safety interlock as a mere property on a Device cannot carry that history or independence; it needs to be a first-class object in its own right.

9.2 Technical Description

`Interlock` (extends `LunexObject` directly — a peer of `Device`, not a subtype of it):

```
Interlock {
  id                : string
  condition         : DeviceRef
  action            : force | block
  target            : DeviceRef
  lockType          : inhibit | lock
  state             : State                (Sub-model 4)
  proofTestHistory  : ProofTest[]
}
```

`Interlock` overrides `methods()` (Sub-model 1 §5.2.1) rather than inheriting the standard Sub-model 4 trigger list: `clear()` — releases a tripped condition once the underlying cause is gone. This is the Interlock's own `clear()`, distinct from the *target* Device's own `clear()` trigger (which moves the target from Inhibited back to Standby once the Interlock permits it).

`lockType` and its target, by the object's current state (this is the corrected, complete rule — see §9.4):

Current state	`lockType: inhibit` targets	`lockType: lock` targets
On	Inhibited (via Emergency Stopping)	Locked (via Emergency Stopping → Inhibited → Disabling → Off → Locking)
Standby	Inhibited (via Inhibiting)	Locked (via Disabling → Off → Locking)
Inhibited	*(already there)*	Locked (via Disabling → Off → Locking)
Off	Locked (via Locking) — Inhibited is unreachable from Off	Locked (via Locking)

Both `lockType` values, when the current state is Off, converge on the same result: **Locked**. This is not two competing behaviors — it follows directly from the Sub-model 4 transition graph having no Off→Inhibited edge.

Safety Instrumented Function (SIF): modeled as an Assembly (Sub-model 3) with `independent: true` and a `SIL` rating as a property of the whole Assembly, not a single Device. Topology is Point-to-Point: Sensor (initiator) → Control Unit (logic solver, typically an SIS) → Actuator (final element). A SIF Assembly must never share a Control Unit with a BPCS Assembly.

`votingArchitecture` — mandatory on every SIF Assembly:

```
SIF Assembly {
  votingArchitecture : {
    sensor         : MooN     (e.g. "2oo3")
    logicSolver    : MooN     (e.g. "1oo1" or "1oo2")
    finalElement   : MooN     (e.g. "1oo2")
  }
}
```

`MooN` is IEC 61508's own voting notation, reused unchanged. Distinct from Sub-model 3's `High-Availability` flag: HA is about uptime and applies to any Assembly, BPCS or SIF; `votingArchitecture` is about SIL integrity — it feeds the PFD calculation and applies only to a SIF. The two are independent and can both be present on the same Assembly for different reasons.

`votingArchitecture` is **mandatory**, not optional like `physicalRef` (Sub-model 1) — the two fields look similar but follow opposite reasoning. Omitting `physicalRef` has a safe, harmless default (the instance is assumed to be its own unique asset). Omitting `votingArchitecture` has no safe default: an implicit `1oo1` would silently assert a SIL-relevant fact — no voting redundancy — that must never be assumed, only explicitly engineered and documented. This is the same principle already governing `AIControlUnit.operatingBounds` (Sub-model 16): mandatory wherever

omission would silently carry a real-world consequence, optional only where omission is genuinely inert.

Stop categories [IEC 60204-1], confirmed against Sub-model 4:

LUNEX state	Category
Controlled Stopping	2 — power remains available
Quick Stopping	1 — controlled, then power removed
Emergency Stopping	0 — immediate power removal

Quick and Controlled Stopping both return to Standby and are never used by an Interlock — only Emergency Stopping (Category 0) is a safety-relevant path from On.

9.3 Application Guidance

1. For every safety function identified in your IEC 61508/61511 SIL assessment, create one `Interlock` instance per condition→target relationship — do not merge multiple independent trip conditions into a single Interlock, or the proof-test history becomes ambiguous.
2. Choose `lockType: inhibit` when the process should pause but logic can stay live (guard doors, process permissives); choose `lockType: lock` when a full lockout is required before restart (machine guarding, maintenance interlocks).
3. Do not assume the target state from `lockType` alone — always resolve it against the table in §9.2 using the target Device's current state.
4. Any SIF must be assigned to its own Assembly at the point of topology design (Sub-model 3), before any shared-Control-Unit decision is made elsewhere in the project.
5. Set `votingArchitecture` for all three layers explicitly, even when a layer is genuinely `1oo1` — an unset layer is a missing engineering decision, not a valid "no vote needed" default. Voting can differ per layer (e.g. `2oo3` sensors feeding a `1oo1` logic solver and a `1oo2` final element) — this is normal, not an inconsistency to resolve.

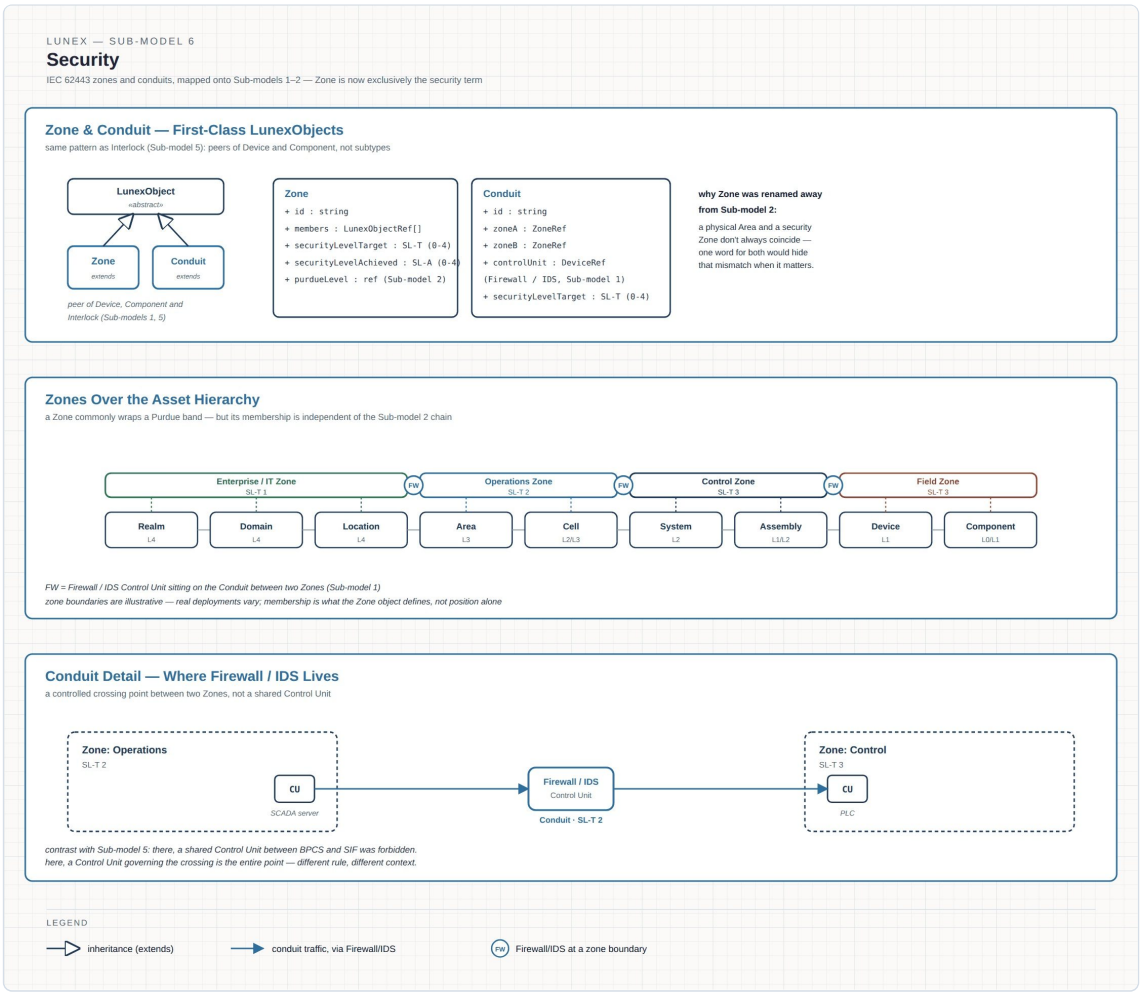
9.4 Note on a corrected edge case

An earlier version of this specification stated that `lockType: inhibit` only applied "from Standby," implying it could not be triggered while an object was Off. This was incomplete: on reflection, when an inhibit-type Interlock condition is active while the target is already Off, the correct behavior is not to silently block the Off→Enabling transition and wait — it is to actively drive the object to Locked, using the same Off→Locking→Locked edge that `lockType: lock` already uses. This is now the specified behavior (§9.2) and applies to every object with an Interlock, not only the `AIControlUnit` case (Sub-model 16) where it was first identified.

---

10. Sub-model 6 — Security

Diagram: `lunex-security-model.svg`



lunex-security-model.svg

10.1 Purpose

IEC 62443 requires zone-and-conduit segmentation with defined Security Levels. As with safety, a security zone needs an identity, membership, and a target/achieved SL — properties that don't fit cleanly as attributes on a Device, and that must not be confused with the physical `Area` concept from Sub-model 2.

10.2 Technical Description

`Zone` (extends `LunexObject`):

```
Zone {
  id                    : string
  members               : LunexObjectRef[]
  securityLevelTarget   : SL-T (0-4)
  securityLevelAchieved : SL-A (0-4)
  purdueLevel           : ref                (Sub-model 2)
}
```

`Conduit` (extends `LunexObject`):

```
Conduit {
  id                    : string
  zoneA                 : ZoneRef
  zoneB                 : ZoneRef
  controlUnit           : DeviceRef          (Firewall / IDS, Sub-model 1)
  securityLevelTarget   : SL-T (0-4)
}
```

A Zone's `members` is independent of the Sub-model 2 containment chain — a Zone commonly wraps a Purdue band (e.g., an "Operations Zone" covering Area/Cell) but its actual boundary is defined by membership, not by position in the hierarchy.

A `Firewall / IDS` (a `Control Unit` subclass, Sub-model 1, marked blue/security) sits at the Conduit boundary between two Zones — this is a normal, intended use of a shared crossing point, distinct from the Sub-model 5 rule that forbids a *shared Control Unit between BPCS and SIF Assemblies*. Both rules concern Control Unit sharing, but in different contexts with opposite intent.

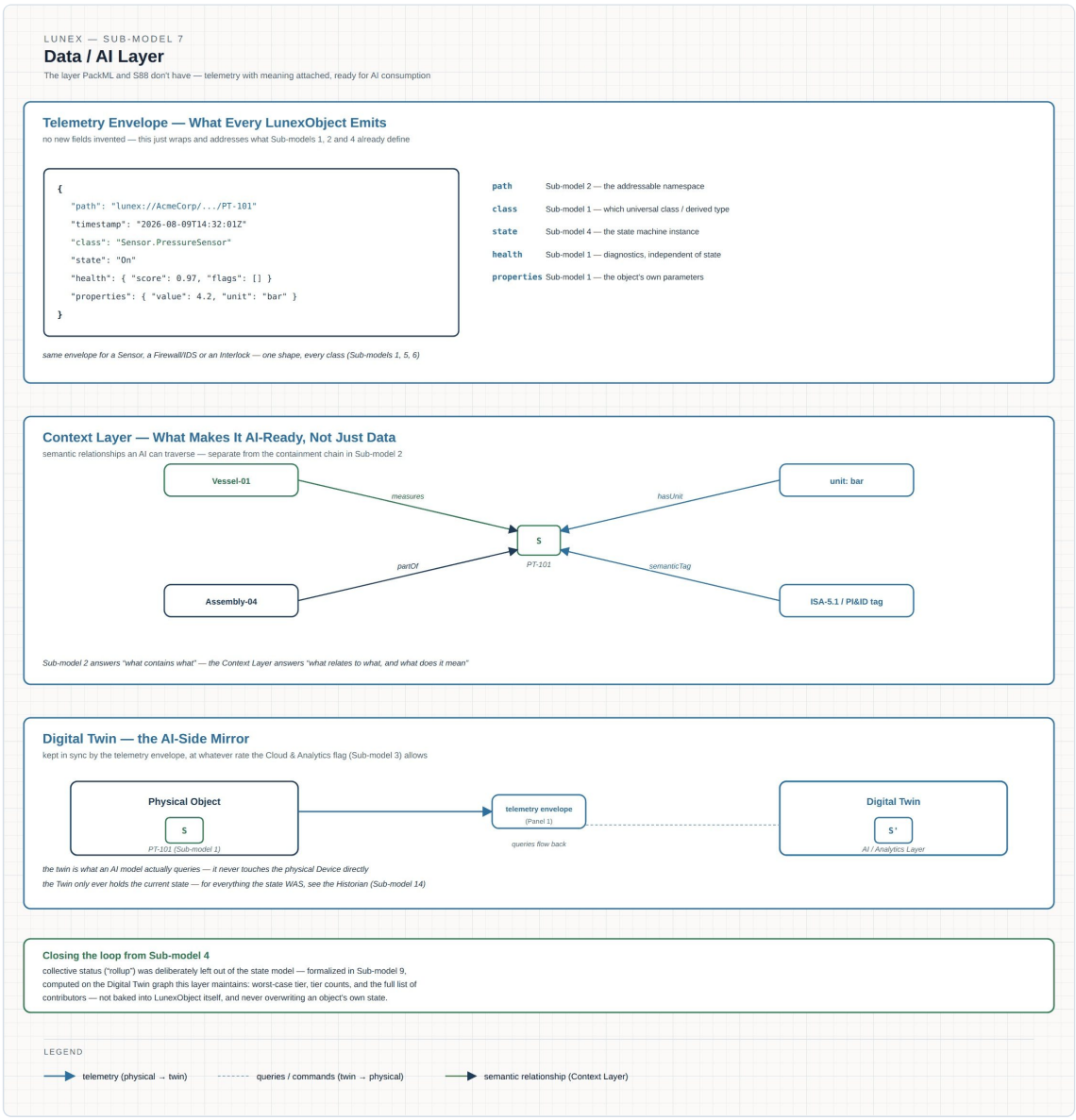
10.3 Application Guidance

1. Perform the IEC 62443 zone-and-conduit exercise first, using standard methodology; then encode its output as `Zone` / `Conduit` instances — LUNEX does not replace that analysis.
2. Do not reuse `Area` (Sub-model 2) as a stand-in for `Zone` even when their boundaries happen to coincide in a given plant — a future re-segmentation of one should not force a re-segmentation of the other.
3. Assign the Firewall/IDS Control Unit to the Conduit's `controlUnit` field, not to either Zone individually.

---

11. Sub-model 7 — Data / AI Layer

Diagram: `lunex-data-ai-model.svg`



lunex-data-ai-model.svg

11.1 Purpose

Neither PackML nor ISA-88 anticipated AI as a structural part of the system. Making an object model “AI-ready” is not simply a matter of exposing raw values over a network — an AI model needs meaning (units, relationships, semantics) attached to data, and a live representation it can query without touching the physical process.

11.2 Technical Description

Telemetry Envelope — no new fields; an addressed, timestamped wrapper around what `LunexObject` already exposes:

```
{
  "path"        : "lunex://.../PT-101"      (Sub-model 2)
  "timestamp"   : ISO 8601
  "class"       : "Sensor.PressureSensor"    (Sub-model 1)
  "state"       : State                       (Sub-model 4)
  "health"      : { score, flags }
  "properties"  : { value, unit, ... }
}
```

The same envelope shape applies to a Sensor, a Firewall/IDS, or an Interlock — one shape, every class.

Context Layer — semantic relationships beyond the containment chain, e.g. `measures`, `partOf`, `hasUnit`, `semanticTag`. Sub-model 2 answers “what contains what”; the Context Layer answers “what relates to what, and what does it mean.”

Digital Twin — the AI-side mirror an AI model actually queries; it never touches the physical Device directly. Kept in sync by the Telemetry Envelope at whatever rate the Sub-model 3 Cloud & Analytics flag allows. The Twin holds only the **current** state — for historical state, see Sub-model 14 (Historian).

Rollup resolution note: collective/rollup status, deferred in Sub-model 4, is computed in this layer (formalized fully in Sub-model 9) — worst-case tier, tier counts, and the full contributor list, computed on the Digital Twin graph this layer maintains, never baked into `LunexObject` itself.

11.3 Application Guidance

1. Implement the Telemetry Envelope as the single outbound message shape from every Control Unit with Cloud & Analytics enabled (Sub-model 3) — do not design a separate schema per object class.
2. Populate Context Layer relationships deliberately during commissioning; they are not inferred automatically from the containment chain.
3. Treat the Digital Twin as read/query-only from the AI side for anything that must reach the physical process — write paths back to the physical Device go through Sub-model 16’s `AIControlUnit`, with its governance intact, not through the Twin directly.

---

12. Sub-model 8 — Naming & Terminology

Diagram: `lunex-naming-model.svg`

LUNEX — SUB-MODEL 8

Naming & Terminology

A living register of every deliberate deviation from S88 / S95 / PackML — and why

Renamed Terms

LUNEX TERM	AVOIDED TERM	REASON	SUB-MODEL
`Realm / Domain / Location`	~~Enterprise / Division / Site~~	broader than an industrial plant; enables the addressable path in Sub-model 7	2
`Area`	~~Zone~~	"Zone" already means something specific in IEC 62443 (Sub-model 6)	2
`Cell`	~~Segment~~	S88 and S95 already use "segment" differently from each other	2
`Point-to-Point`	~~Distributed~~	"Distributed" already means a DCS (multiple controllers) in OT usage	3
`Star`	~~Mixed-Distributed~~	precise, standard network-topology term — needs no explanation	3
`Mesh`	~~Hybrid~~	"Hybrid" already means continuous+batch manufacturing in S88	3
`lockType: inhibit / lock`	~~Process Lock / Machine Lock~~	reuses existing Sub-model 4 state names instead of new, severity-implying labels	5
`Envelope`	~~Container~~	"Container" already means a Docker/Kubernetes deployment unit	7
`Inventory`	~~Library~~	plainer, more approachable — avoids a stuffy academic connotation	1
`Alarm.state (own value space)`	~~reusing Sub-model 4's State enum~~	operator-response states are not object-behavior states — different question, different values	10
`Alarm.origin`	~~reusing Alarm.source's name~~	Alarm.source already means "which Device" (DeviceRef) — origin avoids a same-class field collision	15
`automaticMode: none`	~~on demand~~	manual override is universal, not exclusive to one mode (manualOverrideAvailable)	12/14
`physicalRef: DeviceRef | ComponentRef`	~~physicalRef: DeviceRef only~~	multi-CPU racks need Component granularity — Components already have id (Sub-model 1)	1

Terms Deliberately Kept

renaming is not the default — these already fit LUNEX exactly as written in the source standard

`SIL / SIF`	IEC 61508 / 61511	Sub-model 5
`SL / Zone / Conduit`	IEC 62443	Sub-model 6
`Stop Categories 0 / 1 / 2`	IEC 60204-1	Sub-model 4 / 5
`the state machine itself`	PackML-style pattern	Sub-model 4
`Priority · Shelving · RTN`	ISA-18.2	Sub-model 10 / 11
`System (= Unit)`	ISA-88	Sub-model 2

New Vocabulary

not renamed from anything — these concepts simply don't exist in PackML or S88

`LunexObject` 1 — the abstract base class every object in LUNEX inherits from	`Operator` 1 — resolves OperatorRef — identity + role, deliberately not a permissions system	`Interlock` 5 — a first-class safety object with its own id, state and proof-test history
`Zone` 6 — a security grouping with a target and achieved Security Level	`Conduit` 6 — the governed crossing point between two Zones, where Firewall/IDS sits	`Digital Twin` 7 — the AI-side, queryable mirror of a physical object
`Telemetry Envelope` 7 — the addressed, timestamped wrapper every object streams out	`Context Layer` 7 — semantic relationships an AI can traverse, beyond the containment chain	`Tier / worstTier` 9 — the severity ranking the state machine itself doesn't carry
`Rollup` 9 — cross-object status aggregation, computed in the AI layer, not the object model	`Alarm` 10 — priority = severity × actionability, not severity alone	`Alarm Response Procedure` 11 — reusable, per alarm type — probable cause through corrective action
`GuidanceRecommendation` 12 — simulated scenarios from the Digital Twin, golden scenario optional	`SimulationPolicy` 12 — always / risk-based / none + manualOverrideAvailable — decided per Assembly	`jumpToWorst()` 13 — one action from any node straight to the Device/Alarm causing it
`Historian` 14 — append-only record of every state, ever — the Twin's other destination	`RetentionPolicy` 14 — same shape as SimulationPolicy — always / risk-based / none, by design	`PredictedEvent` 15 — a forward-looking claim, raised into the Alarm system tagged predictive
`ImprovementRecommendation` 15 — always requires human approval — no confidence threshold skips it	`AIControlUnit` 16 — a Control Unit subclass; Interlock forces it exactly like any Device	

The Naming Principle

Rename a term only when it already carries a conflicting meaning elsewhere in the model, or in an adjacent standard LUNEX must interoperate with. Otherwise, reuse established vocabulary — inventing a new word is not innovation if the old one wasn't actually broken.

lunex-naming-model.svg

12.1 Purpose

A model that spans this many pre-existing standards will inevitably face naming collisions — a term two standards use differently, or a term this model itself needs for two different things. Sub-model 8 is not a diagram of a system; it is the living record of every such decision, so future contributors don't have to rediscover the reasoning or accidentally re-introduce a collision that was already resolved.

12.2 Technical Description

Sub-model 8 has no classes of its own. It maintains three registers:

1. **Renamed Terms** — every deliberate deviation from S88/S95/PackML, with the avoided term and the specific reason (e.g. `Realm/Domain/Location` instead of `Enterprise/Division/Site`; `Envelope` instead of `Container`, which already means a Docker/Kubernetes deployment unit; `Alarm.origin` instead of reusing `Alarm.source`'s name, which already means "which Device"; `automaticMode: none` instead of `on-demand`, which wrongly implied manual override was exclusive to one policy mode).
2. **Terms Deliberately Kept** — for balance, every term reused unchanged from a source standard (SIL/SIF, SL/Zone/Conduit, IEC 60204-1 stop categories, ISA-18.2's Priority/Shelving/RTN), so the model does not read as change-for-its-own-sake.
3. **New Vocabulary** — concepts that simply don't exist in PackML or S88 at all (LunexObject, Interlock, Zone, Conduit, Digital Twin, Telemetry Envelope, Context Layer, Tier/worstTier, Rollup, Alarm, Alarm Response Procedure, GuidanceRecommendation, SimulationPolicy, jumpToWorst(), Historian, RetentionPolicy, PredictedEvent, ImprovementRecommendation, AIControlUnit).

The naming principle, which every entry above was tested against:

> *Rename a term only when it already carries a conflicting meaning elsewhere in the model, or in an adjacent standard LUNEX must interoperate with. Otherwise, reuse established vocabulary — inventing a new word is not innovation if the old one wasn't actually broken.*

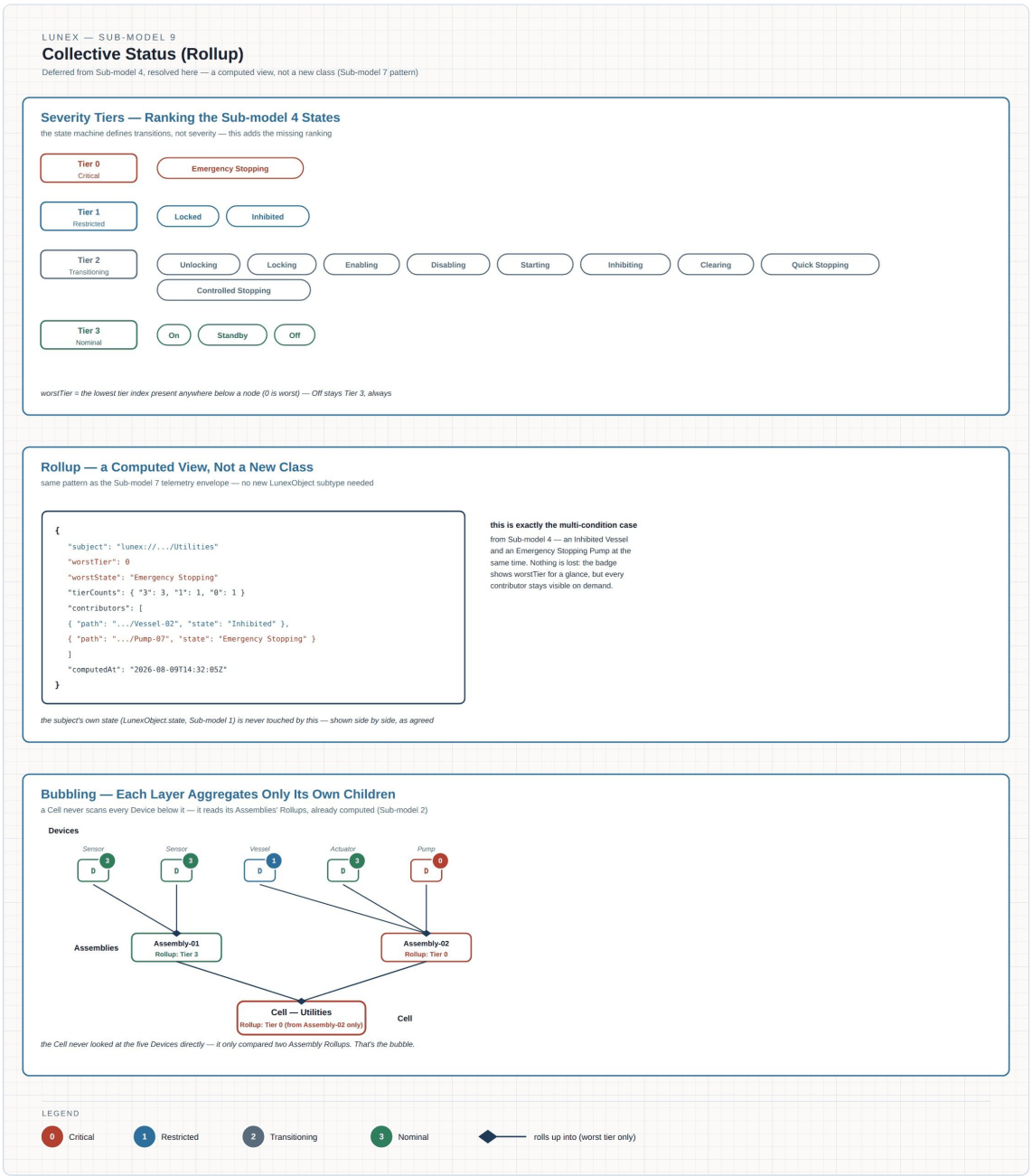
12.3 Application Guidance

1. Before introducing any new term while extending LUNEX, search this register first — the collision may already be documented, or the term you need may already exist under a different, deliberately-chosen name.
2. When a genuine collision is found (as happened between Sub-model 10's `Alarm.source` and Sub-model 15's addition), resolve it by adding an entry here, not by silently picking a name and moving on — the register's value is in being complete, not just correct.
3. Treat "Terms Deliberately Kept" as equally load-bearing as "Renamed Terms" — a proposal to rename SIL, SL, or ISA-18.2's vocabulary should be treated with the same scrutiny as a proposal to *not* rename something that actually collides.

---

13. Sub-model 9 — Collective Status (Rollup)

Diagram: `lunex-rollup-model.svg`



lunex-rollup-model.svg

13.1 Purpose

Sub-model 4 deliberately left one question unanswered: if a Cell has one Device that is Inhibited and another that is Emergency Stopping at the same time, what is the Cell's status? A simple worst-case reduction loses information (which condition, where); scanning the entire tree on every query doesn't scale. Sub-model 9 answers both problems.

13.2 Technical Description

Severity Tiers — a ranking the Sub-model 4 state machine itself does not carry:

Tier	Label	States
0	Critical	Emergency Stopping
1	Restricted	Locked, Inhibited
2	Transitioning	Unlocking, Locking, Enabling, Disabling, Starting, Inhibiting, Clearing, Quick Stopping, Controlled Stopping
3	Nominal	On, Standby, Off

`Rollup` — a computed view, not a stored class:

```
Rollup {
  subject         : LunexObjectRef
  worstTier       : 0-3
  worstState      : State
  tierCounts      : { "3": n, "1": n, "0": n, ... }
  contributors    : [ {path, state}, ... ]
  computedAt      : timestamp
}
```

`worstTier` gives the at-a-glance badge. `tierCounts` and `contributors` preserve every simultaneous condition — this is what resolves the multi-condition case the state machine alone could not: an Inhibited Vessel and an Emergency Stopping Pump are both visible, not collapsed into one value.

Bubbling: a parent's Rollup is computed from its direct children's Rollups only — a Cell reads its Assemblies' Rollups, not every Device beneath them. Each layer aggregates once; nothing scans the full tree.

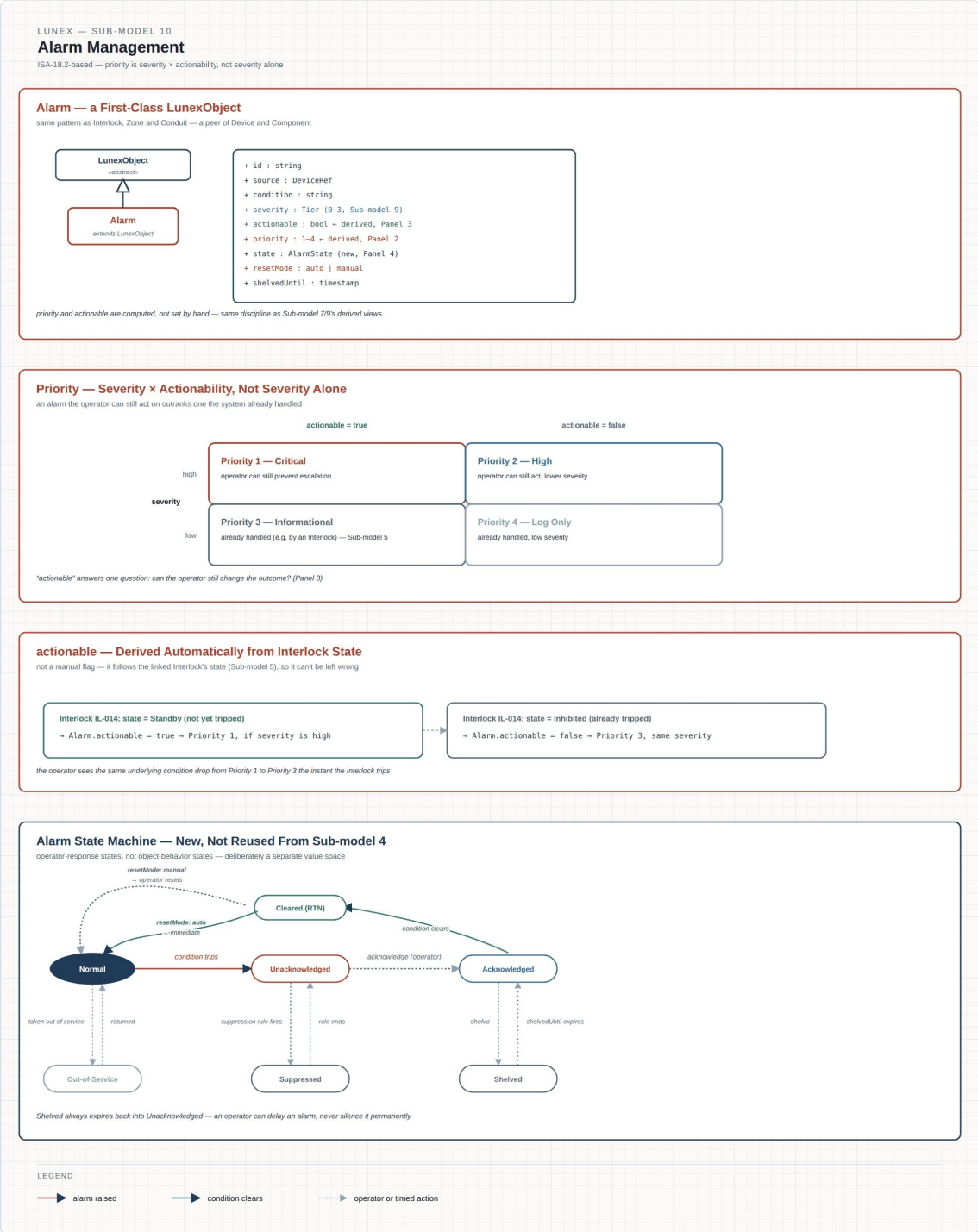
13.3 Application Guidance

1. Compute a Rollup at every level of the Sub-model 2 hierarchy, triggered by a change in any direct child's Rollup or own state — never by polling the full subtree.
2. Surface `worstTier` as the primary visual signal (Sub-model 13); surface `tierCounts` / `contributors` on demand, not by default, to avoid overwhelming an operator screen with detail that isn't yet needed.
3. Do not conflate an object's own `state` (Sub-model 4) with its `Rollup.worstTier` — an object in nominal Standby can still have a Tier 0 Rollup because of a child, and the two values must be shown distinctly (see Sub-model 13).

---

14. Sub-model 10 — Alarm Management

Diagram: `lunex-alarm-model.svg`



lunex-alarm-model.svg

14.1 Purpose

ISA-18.2 defines alarm management principles but does not provide an object model. Sub-model 10 provides one, with a specific correction to a common misconception: severity alone is not priority. An alarm about a condition the system has already handled (say, an Interlock has already tripped) is *less* urgent than an alarm about the same condition while the operator could still prevent it — even though the underlying severity is identical.

14.2 Technical Description

`Alarm` (extends `LunexObject`):

```
Alarm {
  id            : string
  source        : DeviceRef
  origin        : real-time | predictive   (Sub-model 15 — deliberately not named source, see Sub-model 8)
  condition     : string
  severity      : Tier (0-3)              (Sub-model 9)
  actionable    : bool                    (derived, see below)
  priority      : 1-4                     (derived, see below)
  state         : AlarmState              (new value space, see below)
  resetMode     : auto | manual
  shelvedUntil  : timestamp
  procedure     : AlarmResponseProcedureRef | null   (Sub-model 11)
}
```

`actionable` is derived automatically from the linked Interlock's state (Sub-model 5): `true` while the Interlock is Standby (not yet tripped), `false` once it has tripped to Inhibited/Locked. This is never a manually-set flag.

`origin` distinguishes a real-time condition from a forward-looking one; a `PredictedEvent` (Sub-model 15) raises an Alarm with `origin: predictive`, visibly tagged `PREDICTIVE`, joining the same prioritization and screen as any other alarm. `procedure` references the `AlarmResponseProcedure` (Sub-model 11) for this alarm's condition type — `null` is valid for Priority 3/4 alarms, where an ARP is optional (Sub-model 11 §15.2).

Priority matrix — severity × actionability, not severity alone:

	Actionable = true	Actionable = false
High severity	Priority 1 — Critical	Priority 3 — Informational
Low severity	Priority 2 — High	Priority 4 — Log Only

Alarm State Machine — deliberately a *new*, separate value space, not reused from Sub-model 4 (operator-response states answer a different question than object-behavior states):

States: Normal, Unacknowledged, Acknowledged, Shelved, Suppressed, Out-of-Service, Cleared (RTN).

- `condition trips` → Normal → Unacknowledged (automatic)
- `acknowledge` → Unacknowledged → Acknowledged (operator action)
- `condition clears` → Acknowledged → Cleared (RTN) (automatic)
- From Cleared (RTN): if `resetMode: auto`, immediately → Normal; if `resetMode: manual`, requires an explicit `operator resets` action.
- `shelve` (operator action) → Shelved; always expires back to Unacknowledged via `shelvedUntil` — shelving can delay an alarm, never permanently silence it.
- Suppression and Out-of-Service follow the same pattern: entered and exited by explicit, timed, or rule-based actions, never a silent, permanent state.

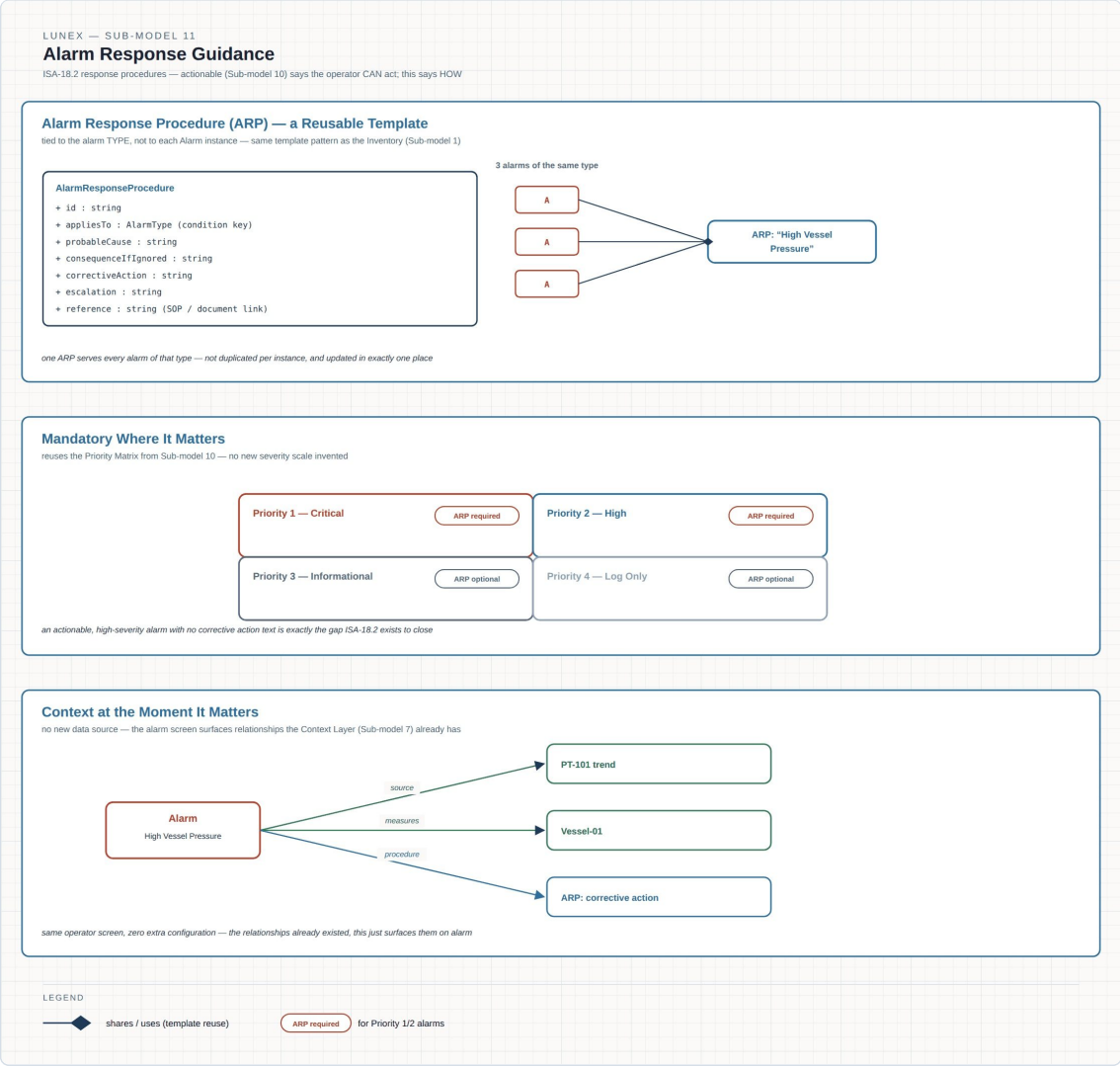
14.3 Application Guidance

1. Set `resetMode: manual` for any alarm where an operator must consciously confirm an abnormal condition occurred, even after it self-clears (typical for safety-adjacent alarms); use `auto` for routine, low-consequence alarms.
2. Never treat `severity` alone as a proxy for `priority` in alarm rationalization work — always compute `actionable` first.
3. Configure `shelvedUntil` with a bounded, sensible expiry for every alarm class — an alarm with no expiry is a design error, not a valid configuration.

---

15. Sub-model 11 — Alarm Response Guidance

Diagram: `lunex-arp-model.svg`



lunex-arp-model.svg

15.1 Purpose

ISA-18.2 calls for documented response procedures, but leaves the “how” open. Sub-model 11 makes response guidance a reusable template rather than freetext duplicated per alarm instance, and ties its context-surfacing directly to what Sub-model 7 already knows — no second system to maintain.

15.2 Technical Description

`AlarmResponseProcedure` **(ARP)** — a reusable template tied to the alarm *type* (condition key), not to each Alarm instance:

```
AlarmResponseProcedure {
  id                    : string
  appliesTo             : AlarmType            (condition key)
  probableCause         : string
  consequenceIfIgnored  : string
  correctiveAction      : string
  escalation            : string
  reference              : string               (SOP / document link)
}
```

`Alarm.procedure` references one `AlarmResponseProcedure`; many Alarm instances of the same type share one ARP — it is updated once, everywhere.

Mandatory coverage, reusing the Sub-model 10 priority matrix directly: an ARP is **required** for every alarm type reachable at Priority 1 or 2; **optional** for Priority 3/4, where there is nothing left to act on.

Context surfacing: when an alarm fires, the operator screen automatically surfaces the relevant Sub-model 7 Context Layer relationships (e.g. the source sensor's trend, the Vessel it `measures`) — this is not a new data source, only a new *view* over data that already exists.

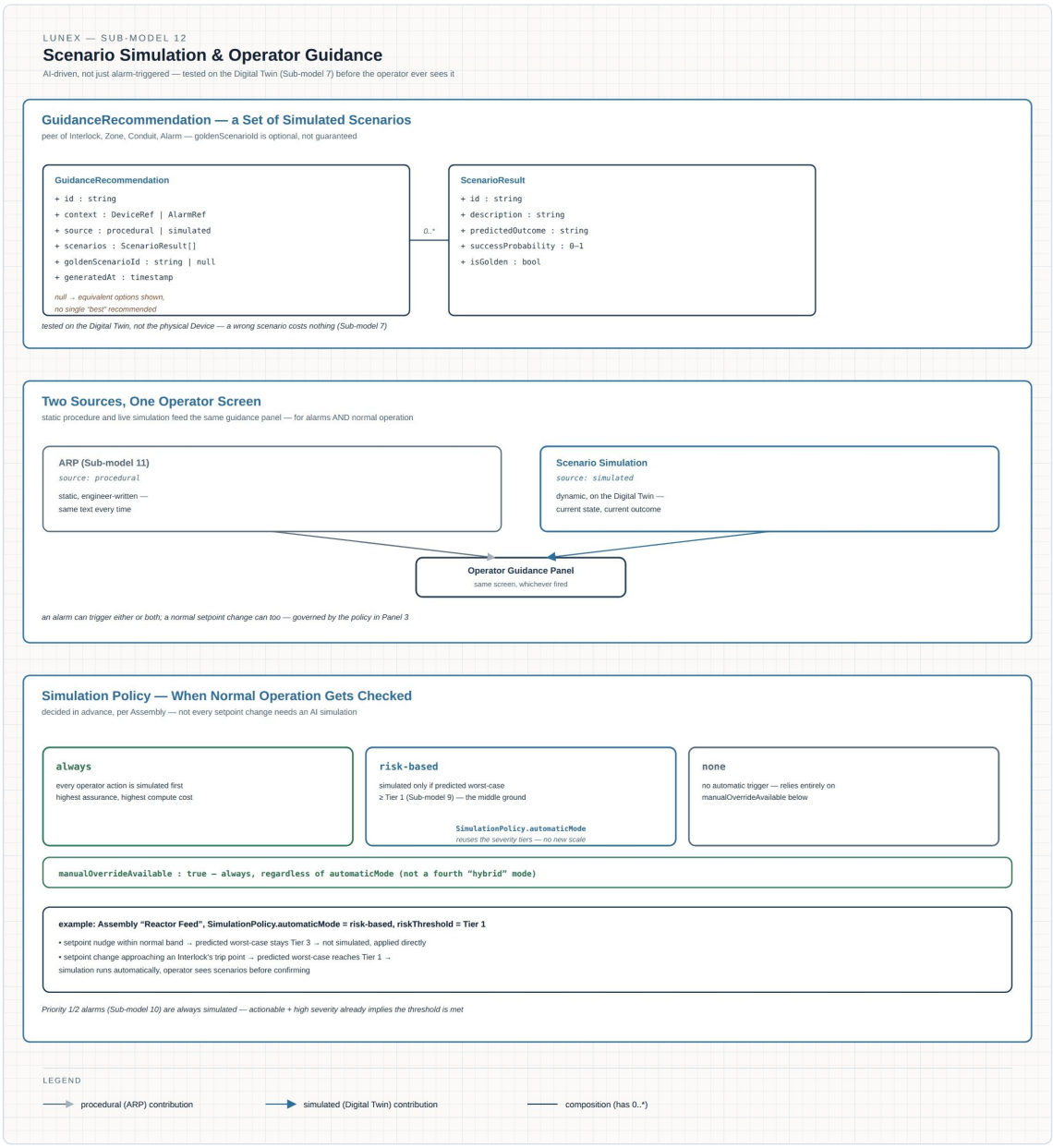
15.3 Application Guidance

1. Author one ARP per distinct alarm *condition*, not per physical instrument — ten identical pressure alarms across ten vessels share one ARP.
2. Treat a missing ARP on a Priority 1/2 alarm as a configuration defect to be fixed before commissioning, not an acceptable gap to fill in later.
3. Ensure the Context Layer relationships an ARP's corrective action depends on (e.g., "check the upstream Vessel's level") are actually populated in Sub-model 7 for that object — an ARP referencing a relationship that was never configured will surface nothing useful at the moment it matters most.

---

16. Sub-model 12 — Scenario Simulation & Operator Guidance

Diagram: `lunex-scenario-model.svg`



lunex-scenario-model.svg

16.1 Purpose

An ARP (Sub-model 11) is static procedural knowledge, written in advance. AI makes something new possible: testing what a specific action would do, right now, on the current process state, before anyone commits to it — and not only for alarms, but for ordinary operation and configuration changes too.

16.2 Technical Description

`GuidanceRecommendation` (extends `LunexObject`; peer of Interlock, Zone, Conduit, Alarm):

```
GuidanceRecommendation {
  id                : string
  context           : DeviceRef | AlarmRef
  source            : procedural | simulated | predictive    (third value added in Sub-model 15)
  scenarios         : ScenarioResult[]
  goldenScenarioId  : string | null
  generatedAt       : timestamp
  state             : Open | Applied | Dismissed | Expired
}

ScenarioResult {
  id                   : string
  description          : string
  predictedOutcome     : string
  successProbability   : 0-1
  isGolden             : bool
}
```

`goldenScenarioId` may be `null` — equivalent options are a valid, honest outcome; the model does not force a single "best" recommendation when the simulation genuinely doesn't produce one.

`state` follows the same shape as `PredictedEvent` (Sub-model 15): `GuidanceRecommendation` overrides `methods()` (Sub-model 1 §5.2.1) with `apply()` and `dismiss()`, both valid only from `Open`. `Expired` is never reached by a method call — a recommendation was generated against the process state *at that moment* (Sub-model 7's Digital Twin), and once enough time passes or the underlying context has visibly moved on, it expires automatically rather than staying presentable as if it were still current.

Scenarios are tested on the Digital Twin (Sub-model 7), never the physical Device — a wrong scenario costs nothing.

`SimulationPolicy` — decided in advance, per Assembly, governing when *normal* operation (not just alarms) gets checked:

```
SimulationPolicy {
  automaticMode           : always | risk-based | none
  riskThreshold           : Tier (0-3)     (used when automaticMode: risk-based)
  manualOverrideAvailable  : true           (always, regardless of automaticMode)
}
```

`manualOverrideAvailable` is not a fourth "hybrid" mode — manual triggering is universal and sits alongside, not instead of, the automatic modes. Priority 1/2 alarms (Sub-model 10) are always simulated, since actionable + high severity already implies the risk threshold is met.

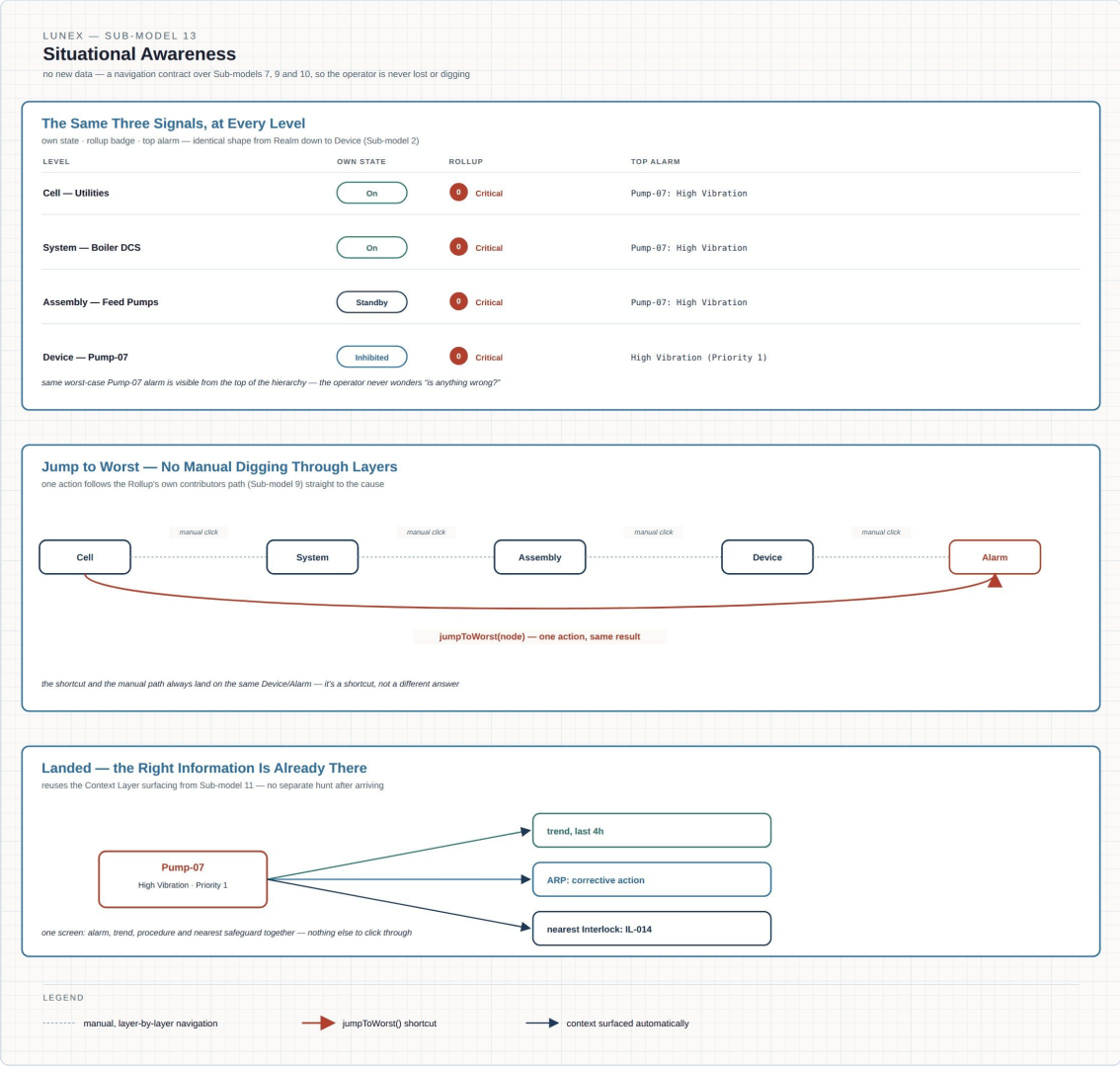
16.3 Application Guidance

1. Set `automaticMode: risk-based` as the default for most Assemblies — `always` is reserved for the highest-consequence processes where compute cost is a non-issue; `none` for low-risk Assemblies where manual override is sufficient.
2. Never assume `goldenScenarioId` is populated in operator-facing UI — always handle the `null` case by presenting equivalent options clearly, not by arbitrarily picking one.
3. Route both `procedural` and `simulated` guidance to the same Operator Guidance Panel — do not build separate interfaces per source; the `source` field is what tells the operator which kind of claim they're looking at.

---

17. Sub-model 13 — Situational Awareness

Diagram: `lunex-sa-model.svg`



lunex-sa-model.svg

17.1 Purpose

Rollup (Sub-model 9), priority (Sub-model 10), and Context (Sub-model 7) each solve part of “does the operator know what’s going on” — but without a navigation contract tying them together consistently at every level, an operator still has to learn a different mental model at each layer of the hierarchy, and still has to click through every intermediate level manually to find the actual cause of a problem.

17.2 Technical Description

Sub-model 13 introduces no new stored data — it is a **navigation and presentation contract** over Sub-models 7, 9, and 10.

The same three signals at every level, from Realm down to Device, identical in shape:

1. Own state (Sub-model 4)
2. Rollup badge — `worstTier` + count (Sub-model 9)
3. Top alarm — highest-priority active alarm at or below this level (Sub-model 10)

`jumpToWorst(node)` — a function, not stored data: follows a Rollup’s own `contributors` path from any node directly to the Device/Alarm responsible for the worst status, in one action. It always lands on the same object the manual layer-by-layer path would reach — it is a shortcut, never a different answer.

Landed context: once navigation reaches the actual cause, the relevant Context Layer relationships and ARP are already surfaced (reusing Sub-model 11's pattern) — no further manual search is required.

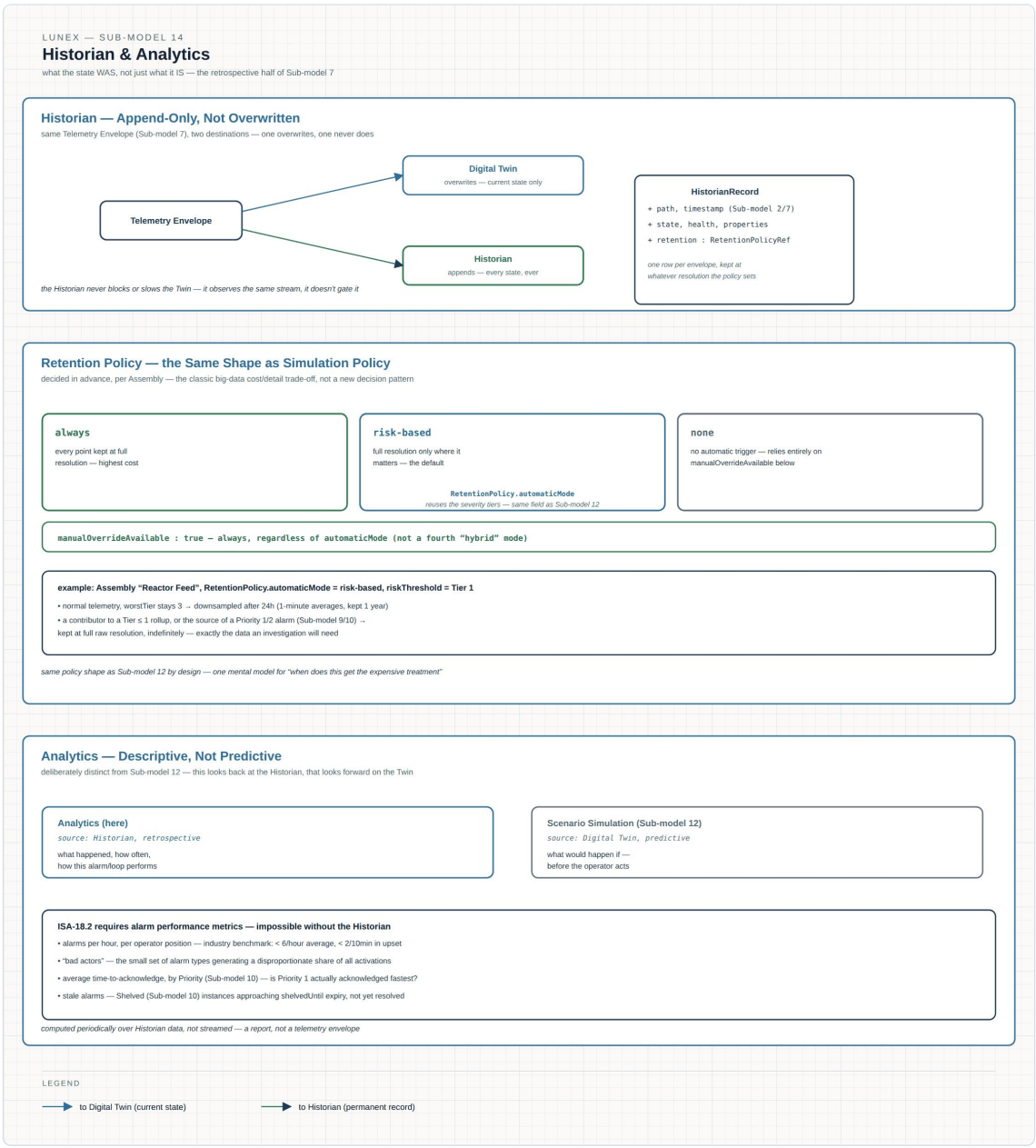
17.3 Application Guidance

1. Implement the three-signal pattern identically at every hierarchy level in the operator UI — resist the temptation to show "more detail" at lower levels and "just a summary" at higher levels using different widgets; consistency of shape is the point.
2. Implement `jumpToWorst()` as a literal traversal of `Rollup.contributors`, not a separate query — this guarantees the shortcut and the manual path can never disagree.
3. Do not add additional context-gathering logic in Sub-model 13 itself; if the right information isn't appearing on arrival, the gap is in Sub-model 7's Context Layer or Sub-model 11's ARP coverage, not in the navigation layer.

---

18. Sub-model 14 — Historian & Analytics

Diagram: `lunex-historian-model.svg`



lunex-historian-model.svg

18.1 Purpose

Sub-model 7 covers what the state *is*; nothing in the model up to this point covers what the state *was*. Without a historical record, ISA-18.2's required alarm performance metrics cannot be computed, no trend is available to underpin Sub-model 15's predictions, and no investigation after an incident has anything to examine.

18.2 Technical Description

Historian — an append-only record of every Telemetry Envelope (Sub-model 7) ever emitted, observing the same stream as the Digital Twin without gating or slowing it. Where the Twin overwrites (current state only), the Historian appends (every state, ever).

```
HistorianRecord {
  path, timestamp    (Sub-model 2/7)
  state, health, properties
  retention          : RetentionPolicyRef
}
```

`RetentionPolicy` — deliberately the same shape as `SimulationPolicy` (Sub-model 12): the classic big-data cost/detail trade-off, resolved with one mental model rather than a new one.

```
RetentionPolicy {
  automaticMode            : always | risk-based | none
  riskThreshold            : Tier (0-3)
  manualOverrideAvailable  : true
}
```

Example: with `automaticMode: risk-based` and `riskThreshold: Tier 1` — normal telemetry (worstTier stays 3) is downsampled after 24h to 1-minute averages, kept 1 year; a contributor to a Tier ≤1 Rollup, or the source of a Priority 1/2 alarm, is kept at full raw resolution indefinitely.

Analytics — deliberately descriptive/retrospective, contrasted with Sub-model 12's predictive simulation: what happened, how often, how a given alarm or loop performs, computed periodically over Historian data (a report, not a telemetry stream).

ISA-18.2 alarm performance metrics, computed here because they are impossible without the Historian: - Alarms per hour, per operator position (industry benchmark: <6/hour average, <2/10min during a process upset) - "Bad actors" — the small set of alarm types generating a disproportionate share of all activations - Average time-to-acknowledge, by Priority (Sub-model 10) - Stale alarms — Shelved instances approaching `shelvedUntil` expiry, still unresolved

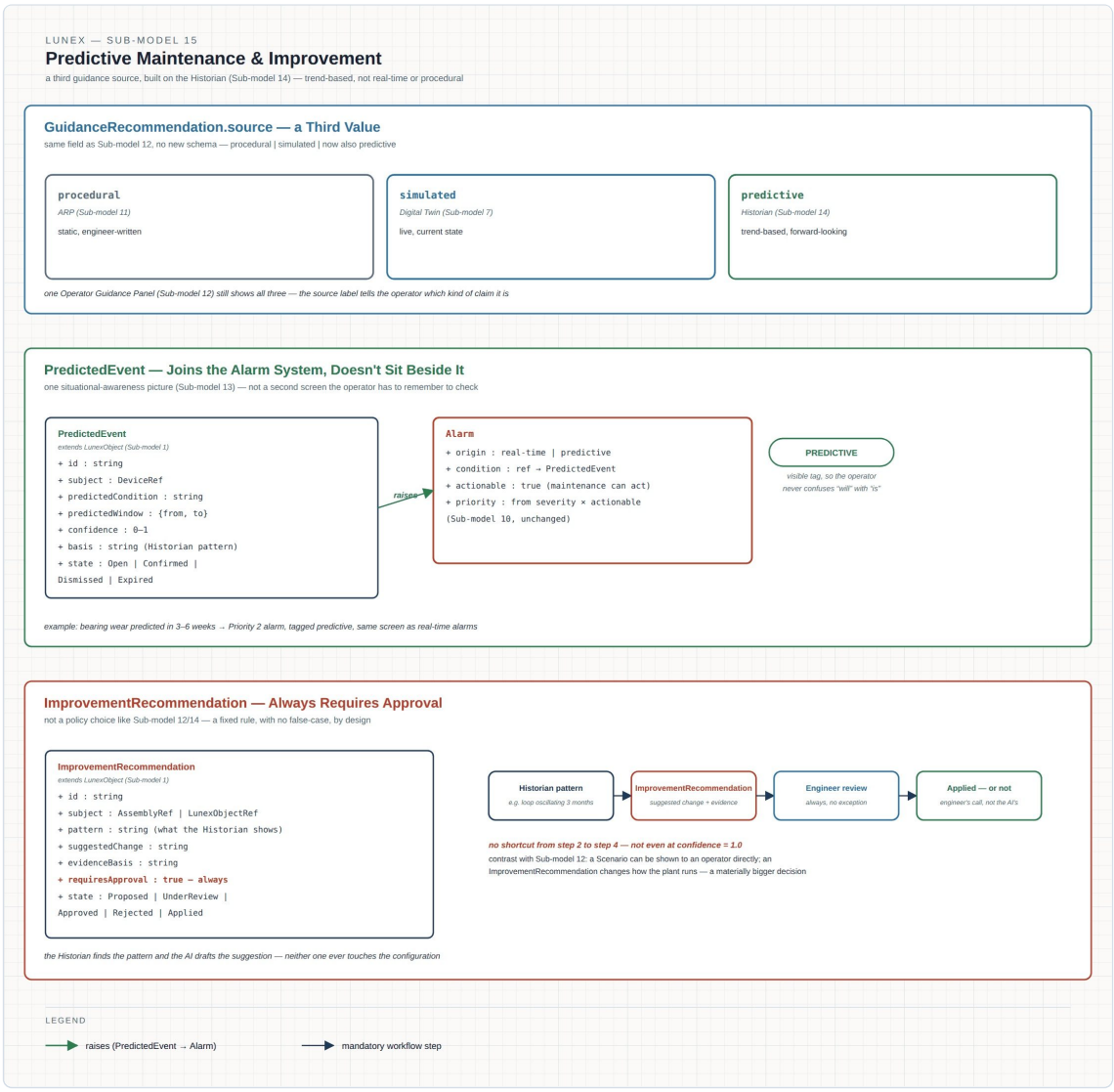
18.3 Application Guidance

1. Set `RetentionPolicy` per Assembly using the same risk-based defaults recommended for `SimulationPolicy` (Sub-model 12) — the two should usually be configured together, not independently.
2. Run the ISA-18.2 performance metrics on a fixed schedule (e.g., weekly/monthly), not on demand only — trend visibility over time is the point.
3. Never write Analytics output back into the Digital Twin — it is a separate, retrospective product, consumed by engineers and Sub-model 15, not by the real-time operator picture.

---

19. Sub-model 15 — Predictive Maintenance & Improvement

Diagram: `lunex-predictive-model.svg`



lunex-predictive-model.svg

19.1 Purpose

Once a Historian exists, trend-based prediction becomes possible — not just “what is happening” or “what would happen if,” but “what is likely to happen, unprompted.” This introduces two distinct needs: surfacing a prediction to an operator without creating a second alarm-like system to monitor, and turning a detected pattern into a suggested configuration change without ever letting AI confidence substitute for human judgment.

19.2 Technical Description

`GuidanceRecommendation.source` gains a third value: `predictive`, built on the Historian (Sub-model 14) rather than the live Digital Twin or a static procedure.

`PredictedEvent` (extends `LunexObject`; peer of Interlock, Alarm, etc.):

```
PredictedEvent {
  id                    : string
  subject               : DeviceRef
  predictedCondition     : string
  predictedWindow        : { from, to }
  confidence              : 0-1
  basis                   : string                (Historian pattern)
  state                    : Open | Confirmed | Dismissed | Expired
}
```

A `PredictedEvent` **raises** into the ordinary Sub-model 10 `Alarm` system — `Alarm.origin: predictive`, visibly tagged `PREDICTIVE` — rather than occupying a separate maintenance-planning screen the operator must remember to check separately (this joins Sub-model 13's single situational-awareness picture, it doesn't sit beside it). `Alarm.origin` is deliberately not named `Alarm.source`, since that field already means "which Device" on the same class (see Sub-model 8).

`state: Expired` matters specifically for Sub-model 14's Analytics: a prediction whose window passed without confirmation is exactly the data needed to evaluate how reliable predictions actually are. `PredictedEvent` overrides `methods()` (Sub-model 1 §5.2.1) with `confirm()` and `dismiss()`, both only valid from `Open`; `Expired` is never reached by a method call, only by the window passing unconfirmed.

`ImprovementRecommendation` (extends `LunexObject`):

```
ImprovementRecommendation {
  id                  : string
  subject             : AssemblyRef | LunexObjectRef
  pattern             : string             (what the Historian shows)
  suggestedChange     : string
  evidenceBasis       : string
  requiresApproval    : true                (always — no false case)
  state               : Proposed | UnderReview | Approved | Rejected | Applied
}
```

`ImprovementRecommendation` overrides `methods()` with `startReview()`, `approve()` / `reject()`, and `apply()` — one method per arrow in the workflow below, with no method that skips a step.

Mandatory workflow, no shortcut: Historian pattern → `ImprovementRecommendation` drafted → engineer review (always, no exception) → Applied or not, at the engineer's discretion — never the AI's. This holds even at `confidence: 1.0`. The distinction from Sub-model 12: a `ScenarioResult` can be shown to an operator directly because it only informs one decision in the moment; an `ImprovementRecommendation` changes how the plant runs going forward — a materially bigger decision that always requires human approval.

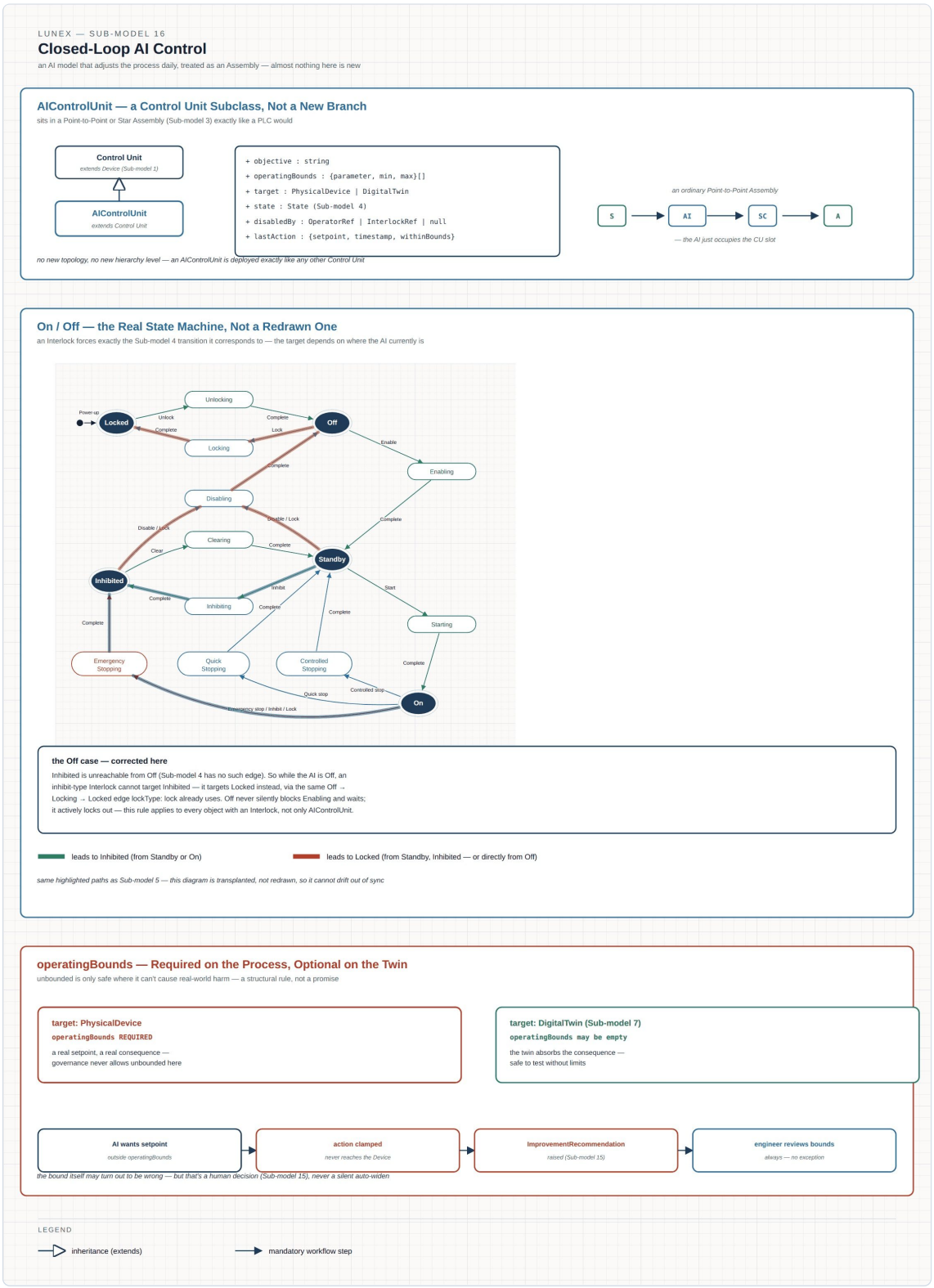
19.3 Application Guidance

1. Route every `PredictedEvent` through the same Alarm prioritization logic as real-time alarms (Sub-model 10) — do not build a separate priority scheme for predictions.
2. Never expose a mechanism to auto-apply an `ImprovementRecommendation`, regardless of confidence — `requiresApproval` is a fixed rule in this specification, not a configurable default.
3. Use `PredictedEvent.state: Expired` data as direct input to a running accuracy assessment of the prediction model generating them — a model that is frequently wrong should be visible as such, not silently trusted.

---

20. Sub-model 16 — Closed-Loop AI Control

Diagram: `lunex-ai-control-model.svg`



lunex-ai-control-model.svg

20.1 Purpose

Some AI models don't just recommend — they adjust the process directly, continuously, without a human in the loop for every action. This is qualitatively different from Sub-model 15's `ImprovementRecommendation` (a one-time, approved configuration change) and needs its own governance: the ability to enable/disable the AI like any other Device, and a hard boundary on what it is allowed to do autonomously.

20.2 Technical Description

`AIControlUnit` (extends `Control Unit`, Sub-model 1 — not a new branch):

```
AIControlUnit extends ControlUnit {
  objective         : string
  operatingBounds   : { parameter, min, max }[]
  target            : PhysicalDevice | DigitalTwin
  state             : State                          (Sub-model 4, unchanged)
  disabledBy        : OperatorRef | InterlockRef | null
  lastAction        : { setpoint, timestamp, withinBounds }
}
```

`target` here means something different from `Interlock.target` (Sub-model 5) — an Interlock's `target` is the Device it forces a transition on; an AIControlUnit's `target` is what its output is routed to. The two never collide in practice, since no object is ever both classes at once; this is the same pattern already established by `state`, which means something different on `Alarm` than on most other classes (Sub-model 8). `disabledBy` resolves `OperatorRef` to a first-class `Operator` (Sub-model 1 §5.2).

It occupies an ordinary Control Unit slot within a Point-to-Point or Star Assembly (Sub-model 3) — no new topology shape is required.

On/Off reuses the Sub-model 4 state machine, unmodified. An Interlock (Sub-model 5) forces an `AIControlUnit` to Standby or Locked exactly as it would any other Device — including the corrected Off→Locked behavior (§9.4): from Off, an inhibit-type Interlock targets Locked directly, not Inhibited. Consequently, `AIControlUnit` does **not** override `methods()` (Sub-model 1 §5.2.1): `enable()` / `disable()` / `start()` / `emergencyStop()` and the rest are exactly the standard Sub-model 4 list, unlike `Alarm`, `Interlock`, `GuidanceRecommendation`, `PredictedEvent`, or `ImprovementRecommendation`, each of which defines its own trigger set. An AI performing closed-loop control is switched on and off the same way any Control Unit is.

`operatingBounds` governance:

`target`	`operatingBounds`
PhysicalDevice	**Required.** A real setpoint has a real consequence; governance never allows unbounded operation here.
DigitalTwin (Sub-model 7)	**May be empty.** The Twin absorbs the consequence — safe to test without limits.

An action outside `operatingBounds` is **clamped** — it never reaches the physical Device — and separately **raises an `ImprovementRecommendation`** (Sub-model 15) suggesting the bound itself be reviewed. The bound may turn out to be wrong, but that is always a human decision, never a silent auto-widening.

20.3 Application Guidance

1. Never deploy an `AIControlUnit` with `target: PhysicalDevice` and empty `operatingBounds` — this specification treats that configuration as invalid, not merely discouraged.
2. Use `target: DigitalTwin` for all model testing and tuning before any production deployment against a physical target.
3. Wire at least one Interlock to every production `AIControlUnit`, exactly as you would for any other Control Unit performing a comparable function — an AI performing closed-loop control is not exempt from the safety model.
4. Treat repeated out-of-bounds attempts (visible via `ImprovementRecommendation` frequency) as a signal to review whether the objective or the bounds are miscalibrated — not as a reason to widen the bounds reflexively.

---

21. References

- ISA-88 (IEC 61512) — Batch Control
- ISA-95 (IEC 62264) — Enterprise-Control System Integration
- ISA-TR88.00.02 (PackML) — Machine and Unit States
- IEC 61508 — Functional Safety of Electrical/Electronic/Programmable Electronic Safety-Related Systems
- IEC 61511 — Functional Safety: Safety Instrumented Systems for the Process Industry Sector
- IEC 62443 — Industrial Communication Networks: Network and System Security
- ISA-18.2 — Management of Alarm Systems for the Process Industries
- IEC 60204-1 — Safety of Machinery: Electrical Equipment of Machines
- Purdue Enterprise Reference Architecture (PERA)

LUNEX is an independent work and is not affiliated with, endorsed by, or a replacement for any of the standards bodies listed above.

This document is part of the LUNEX project. See `README.md` and `LICENSE` in the repository root, and lunex.cloud.

Appendix A — Schema Reference

Every class defined across the sixteen sub-models, collected here for lookup without paging back through each chapter. Attributes shown are exactly as specified in the relevant Sub-model section; this appendix adds nothing new; it only re-collects.

Sub-model 1 — Object / Class Model

```
LunexObject {                          (abstract base — every object has this)
  id            : string
  tag           : string
  class         : Class
  parent        : LunexObject
  state         : State                 (Sub-model 4)
  health        : Health                { score: 0-100, flags: string[] }
  properties    : Map<Key, Value>
  methods()     : Procedure[]           (derived from state — Sub-model 1 §5.2.1)
}

Device extends LunexObject {           (abstract)
  physicalRef : DeviceRef | ComponentRef | null
}

Component extends LunexObject {}       (abstract — not independently addressable)

Sensor, Transducer, Control Unit, Signal Converter, Actuator
  all extend Device, no additional attributes — distinguished by
  function-interface (Sensing / Converting / Controlling / Actuating)

Operator extends LunexObject {
  role : string   (e.g. "Control Room Operator", "Safety Engineer")
}
```

Sub-model 5 — Safety

```
Interlock extends LunexObject {
  condition          : DeviceRef
  action             : force | block
  target             : DeviceRef
  lockType           : inhibit | lock
  proofTestHistory   : ProofTest[]
}

SIF Assembly {
  independent          : true
  votingArchitecture    : { sensor: MooN, logicSolver: MooN, finalElement: MooN }
}
```

Sub-model 6 — Security

```
Zone extends LunexObject {
  members                 : LunexObjectRef[]
  securityLevelTarget     : SL-T (0-4)
  securityLevelAchieved   : SL-A (0-4)
  purdueLevel             : ref               (Sub-model 2)
}

Conduit extends LunexObject {
  zoneA                   : ZoneRef
  zoneB                   : ZoneRef
  controlUnit             : DeviceRef          (Firewall / IDS)
  securityLevelTarget     : SL-T (0-4)
}
```

Sub-model 9 — Collective Status

```
Rollup {                              (computed view, not a stored class)
  subject         : LunexObjectRef
  worstTier       : 0-3
  worstState      : State
  tierCounts      : { "3": n, "1": n, "0": n, ... }
  contributors    : [ {path, state}, ... ]
  computedAt      : timestamp
}
```

Sub-model 10 — Alarm Management

```
Alarm extends LunexObject {
  source        : DeviceRef
  origin        : real-time | predictive  (Sub-model 15)
  condition     : string
  severity      : Tier (0-3)              (Sub-model 9)
  actionable    : bool                    (derived from linked Interlock's state)
  priority      : 1-4                     (derived: severity × actionable)
  state         : AlarmState              (own value space — Sub-model 10 §14.2)
  resetMode     : auto | manual
  shelvedUntil  : timestamp
  procedure     : AlarmResponseProcedureRef | null   (Sub-model 11)
}
```

Sub-model 11 — Alarm Response Guidance

```
AlarmResponseProcedure {
  appliesTo               : AlarmType            (condition key)
  probableCause           : string
  consequenceIfIgnored    : string
  correctiveAction        : string
  escalation              : string
  reference                : string               (SOP / document link)
}
```

Sub-model 12 — Scenario Simulation & Operator Guidance

```
GuidanceRecommendation extends LunexObject {
  context            : DeviceRef | AlarmRef
  source             : procedural | simulated | predictive
  scenarios          : ScenarioResult[]
  goldenScenarioId   : string | null
  generatedAt        : timestamp
  state              : Open | Applied | Dismissed | Expired
}

ScenarioResult {
  description           : string
  predictedOutcome      : string
  successProbability    : 0-1
  isGolden              : bool
}

SimulationPolicy {
  automaticMode            : always | risk-based | none
  riskThreshold            : Tier (0-3)     (used when automaticMode: risk-based)
  manualOverrideAvailable  : true            (always)
}
```

Sub-model 14 — Historian & Analytics

```
HistorianRecord {
  path, timestamp    (Sub-model 2/7)
  state, health, properties
  retention          : RetentionPolicyRef
}

RetentionPolicy {
  automaticMode            : always | risk-based | none
  riskThreshold            : Tier (0-3)
  manualOverrideAvailable  : true
}
```

Sub-model 15 — Predictive Maintenance & Improvement

```
PredictedEvent extends LunexObject {
  subject               : DeviceRef
  predictedCondition     : string
  predictedWindow        : { from, to }
  confidence              : 0-1
  basis                   : string               (Historian pattern)
  state                    : Open | Confirmed | Dismissed | Expired
}

ImprovementRecommendation extends LunexObject {
  subject             : AssemblyRef | LunexObjectRef
  pattern             : string             (what the Historian shows)
  suggestedChange     : string
  evidenceBasis       : string
  requiresApproval    : true                (always — no false case)
  state               : Proposed | UnderReview | Approved | Rejected | Applied
}
```

Sub-model 16 — Closed-Loop AI Control

```
AIControlUnit extends ControlUnit {
  objective         : string
  operatingBounds   : { parameter, min, max }[]
  target            : PhysicalDevice | DigitalTwin
  disabledBy        : OperatorRef | InterlockRef | null
  lastAction        : { setpoint, timestamp, withinBounds }
}
```

`methods()` **quick reference** (Sub-model 1 §5.2.1)

State	Triggers
`Locked`	`unlock()`
`Off`	`lock()`, `enable()`
`Standby`	`start()`, `disable()`, `inhibit()`
`Inhibited`	`clear()`, `disable()`
`On`	`controlledStop()`, `quickStop()`, `emergencyStop()`
any transient state	*(empty)*
`Alarm` (override)	`acknowledge()`, `shelve()`, `operatorReset()`, `suppress()` / `unsuppress()`, `takeOutOfService()` / `returnToService()`
`Interlock` (override)	`clear()`
`GuidanceRecommendation` (override)	`apply()`, `dismiss()`
`PredictedEvent` (override)	`confirm()`, `dismiss()`
`ImprovementRecommendation` (override)	`startReview()`, `approve()`, `reject()`, `apply()`