Unified Device Integration Architecture for EtherNet/IP: Stateless, Event-Driven Configuration and Control

Michael Miller
Senior System Architect
Rockwell Automation, Inc.

Paul Brooks
Senior Manager, Open Architecture Management
Rockwell Automation, Inc.

Presented at the ODVA
2026 Industry Conference & 24th Annual Meeting of Members
October 19-22, 2026
Vilamoura, Algarve, Portugal

I. Abstract

EtherNet/IP devices are becoming more complex, and the associated device configuration in modern automation systems must balance rich, device-specific behavior with cloud-scale deployment, interoperability, and predictable host integration. The paper will describe an approach that combines stateless business logic, schema-versioned APIs to address this challenge. In this model, device vendor-supplied applications remain stateless with interactions where the data models are provided through asynchronous events, while controller configuration applications own project state, sequencing, and consistency boundaries. Operational workflows are exchanged as CloudEvents over MQTT using versioned schemas, with explicit separation between transport responsibilities and profile logic. Enabling moving communications to the Edge, CIP direct-to-device operations use correlated request/response tokens and repeatable transport contracts, enabling deterministic execution for one-shot and polling scenarios without introducing cross-request state leakage in services.

In parallel, the proposed solution preserves compatibility with established artifacts and practices, including static descriptors such as EDS and EDDL-derived structures, while introducing dynamic registration and extensible JSON schema contracts for host/device coordination. The resulting architecture is expected to improve interoperability across hosts and device families, reduce integration risk for device software developers, and provide a scalable baseline for more complex configuration, online diagnostics, and multi-step device workflows to be defined by device vendors. This paper provides an overview of the architecture decisions, implementation implications, and ecosystem benefits for vendors, host developers, and end users.

The approach is deliberately additive rather than disruptive. Static descriptors such as EDS files and EDDL-derived structures continue to serve as authoritative sources; dynamic registration and extensible JSON contracts are layered on top of them for host and device coordination. The intended result is better interoperability across hosts and device families, lower integration risk for

 ©2026 ODVA, Inc.

device software developers, and a scalable baseline for the more demanding scenarios, complex configuration, online diagnostics, and multistep device workflows that device vendors increasingly need to express.

Purely data-driven descriptors such as EDS and EDDL work well for simple parameter sets. The difficulty is that modeling every device with a single, static schema follows a steep complexity curve. As devices add conditional parameter visibility, cross-parameter dependencies, multi-step calibration workflows, and vendor-specific business rules, the descriptor grammar must grow without bound, and every host must implement an ever-expanding interpreter to keep pace. This is the structural cause behind the fragmentation and cost documented in Section II: each new class of device behavior forces coordinated change across every host that claims to support it. In practice this is felt most acutely with EDDL; for EDS it applies mainly in the rarer cases where the descriptor grammar itself must be extended with new sections or keywords.

II. Definition of Terms

Term	Definition
Adapter	A component that translates between TEMPO's CloudEvents contract and another runtime; a host adapter coordinates concurrency, deduplication, and workflow on the host side; a communication adapter or execution adapter applies accepted configuration to a physical device asynchronously; legacy DTM or FDI wrappers expose existing packages through the same API without reimplementation
API-First	An integration approach where you design the API before you write any of the code behind it. The API is the collaboration boundary, implementations stay within each vendor's or standards body's domain, and the contract is expressed as stateless, platform-independent, schema-versioned operations over a message bus
At-least-once delivery	A messaging guarantee in which a broker may deliver the same message more than once; TEMPO pairs this with idempotent processing and attempt-based deduplication so configuration side effects occur exactly once despite duplicate deliveries
Business Logic	The device-specific rules, calculations, and operational behavior (parameter dependencies, methods, and validation) that turn a user's intent and the current device state into a valid configuration; in TEMPO it runs headlessly as stateless pure functions in the TEMPO Service, kept separate from presentation.
Canonical device state	The structured, protocol-agnostic device configuration the host owns as the single source of truth; derived from the event log, merged from TEMPO Service responses, and projected into view models and OPC UA
CIP	Common Industrial Protocol
Client	The presentation layer in TEMPO's four-layer model: it renders view models, sends mutations as CloudEvents, tracks the commit version, and surfaces validation errors from returned device state; desktop tools, mobile apps, web UIs, and AI agents participate as clients through the same contract
CloudEvents	CNCF-standard envelope for describing event data (cloudevents project)
Commit version	A monotonically increasing integer owned by the host that identifies the current accepted configuration; carried on every mutation as an optimistic concurrency precondition and advanced only when returned device state has no validation errors

 ©2026 ODVA, Inc.

Term	Definition
Container	A portable, isolated runtime package (for example Docker/OCI) used to deploy TEMPO Services and legacy DTM wrappers with resource limits; distinct from Edge, which describes where compute runs rather than how it is packaged
CRUD	Create, Read, Update, Delete — the familiar pattern for operating on structured records; in TEMPO the contract is CRUD-ish: create initializes a device configuration, query reads device state and view models, and update (mutate) changes device state through validation runs on the returned state rather than as a separate operation, and there is no standalone delete of individual parameters because configuration evolves through state updates and the event log
Data-driven	An interaction style in which each payload carries self-describing schema metadata (type, range, units, dependencies) alongside values, so a client operates on typed device state without embedding device-specific logic; in TEMPO this is how the CRUD-ish contract is made usable — query and update exchange structured device state and validation errors in state, not opaque blobs or black-box calls
Deterministic	A property of computation in which the same inputs always produce the same outputs; TEMPO requires TEMPO Services to be deterministic so concurrent replicas, event replay, and automated testing converge on identical state
Device state	The structured configuration payload a TEMPO Service receives, validates, and returns; when validation passes and the host merges it, device state becomes part of canonical device state; validation errors are recorded in the returned state itself rather than as a separate error channel
DTM	Device Type Manager (the executable device component of FDT)
EDD / EDDL	Electronic Device Description / its language (IEC 61804)
EDS	Electronic Data Sheet
Edge	Computing deployed close to the physical plant or equipment (gateways, cells, on-prem nodes) rather than in a distant cloud data center; edge deployments favor outbound-only messaging, offline tolerance, and local execution of configuration and control workflows
Event log	The host's append-only record of configuration events from which canonical device state is derived by replay; supports audit, temporal queries, disaster recovery, and change tracking
Event sourcing	A persistence pattern in which authoritative state is reconstructed by replaying an immutable sequence of events rather than overwriting records in place; in TEMPO the host event log is authoritative and canonical state is a fold over that log
FCG	FieldComm Group
FDI	Field Device Integration
FDT	Field Device Tool
Gateway	A protocol bridge that exposes TEMPO's internal MQTT/CloudEvents traffic to external clients over REST, WebSocket, gRPC, Server-Sent Events, or legacy protocols while preserving the CloudEvents envelope end to end; distinct from an edge gateway, which may run host or adapter logic close to the plant
Horizontal scaling	Adding capacity by running more service instances behind a load balancer rather than increasing the resources of a single instance;

Term	Definition
	effective horizontal scaling requires that each instance hold no session-specific state so any replica can handle any request
Host	The service or runtime that owns canonical device configuration and project state; it checks preconditions on incoming mutations, invokes the stateless TEMPO Service, merges returned device state, and typically is the application the user interacts with for configuring devices. This can be a desktop application, a cloud based website or a mobile application.
Idempotency	The property that processing the same logical message more than once has the same effect as processing it once; enforced by correlation identifiers, deduplication windows, and commit-version acceptance rules so at-least-once delivery never double-applies configuration
JSON Patch	RFC 6902 standard for describing differences between two JSON documents; TEMPO uses JSON Patch deltas for change tracking, undo/redo, and dirty-state indicators in the host
JSON Schema	A vocabulary for describing JSON document structure; TEMPO expresses the canonical device model and parameter metadata as JSON Schema augmented with EDDL-compatible constructs where they add value
Margo	A Linux Foundation initiative for interoperable edge-application/device orchestration
Message bus	The asynchronous messaging infrastructure (typically MQTT) over which TEMPO Services exchange CloudEvents; decouples producers and consumers so hosts, clients, TEMPO Services, and adapters participate through a shared contract without synchronous coupling
MQTT	A lightweight publish/subscribe messaging transport
Multi-tenant	A deployment model in which one service instance serves multiple customers or plants, with tenant identity carried in the request envelope and isolation enforced at the service boundary
Mutation	A state-changing client operation sent to the host as a CloudEvent; carries the proposed change and current commit version; distinct from query, which reads state without advancing configuration ODVA the standards body for EtherNet/IP and CIP
Optimistic concurrency	A coordination pattern in which writers include the last known version when submitting changes and the host rejects stale writes before side effects occur; in TEMPO implemented through the commit version
OPC UA	OPC Unified Architecture (IEC 62541)
PoC	Proof of concept. An implementation used to show the design works.
Publish/subscribe	A messaging pattern in which publishers emit events to topics and subscribers receive only the messages they have subscribed to; MQTT's publish/subscribe model supports multi-tenant filtering, fan-out to multiple clients, and outbound-only edge connectivity
Pure function	A computation with no side effects and no dependency on hidden state: given the same request, models, and host context, a TEMPO Service always returns the same outputs; this is the implementation model for TEMPO Services
Sandbox / sandboxing	Isolating untrusted or vendor-supplied code (imported EDS files, legacy DTMs, TEMPO Services) within restricted execution environments so compromised descriptions or binaries cannot access host resources beyond their contract
Semantic versioning	A version-numbering scheme (major.minor.patch) in which major increments signal breaking changes, minor increments add compatible

Term	Definition
	features, and patch increments fix defects; TEMPO uses semantic versioning for TEMPO Service evolution and schema negotiation
SIC	Strategic Integration Committee (formed inside FCG post-merger)
SDO	Standards Developing Organization; an industry body that defines and certifies a field protocol and its device descriptors (e.g., ODVA for EtherNet/IP, FCG for HART and FOUNDATION Fieldbus, PNO for PROFINET). In TEMPO, SDOs collaborate as peers through a shared API contract while each retains ownership of its artifacts, certification labs, and implementations
SOX	Sarbanes-Oxley Act; U.S. legislation mandating financial recordkeeping, internal controls, and audit trails.
SPI	Service Provider Interface (the contract a TEMPO Service satisfies)
Stateless services	Services that hold no session-specific state between requests, so any replica can handle any request; this enables horizontal scaling, rolling deployment, and multi-tenant isolation; in TEMPO, TEMPO Services are stateless while the host owns persistence
TEMPO	Transport-agnostic Event Model for Profile Orchestration. The API-First device integration architecture described here.
TEMPO Framework Library	A standard, reusable wrapper around an EDS (or EDD) that implements the TEMPO API, so a functional TEMPO Service can be generated from an existing descriptor with little or no additional vendor code.
TEMPO Service	The stateless, vendor or SDO supplied application that implements device-specific business logic for a device type; given a request, current models, and host context, it applies changes to device state, validates that state, and returns updated models, view models, and device state with any validation errors recorded in the state itself
UID / UIP	User Interface Description / User Interface Plug-in (FDI)
View model	A UI-oriented projection of device state returned by the TEMPO Service for client rendering; carries page layout, labels, and current values alongside schema metadata so clients remain device-agnostic
Zero-trust	A security model in which no component is trusted by default based on network location or origin; in this architecture, trust is enforced at the message boundary — only signed, schema-validated CloudEvents cross service boundaries, limiting lateral movement if a device container or description is compromised

III. Motivation & Landscape

A. The Industry Challenge

The legacy architecture underneath adds further constraints. EDS is a mature, well understood descriptor whose evolution ODVA has actively guided (Blair, *A Modern Approach to CIP Device Descriptions*, 2015 ODVA Industry Conference; Frazer & Snide, *Current Developments in xDS Device Descriptions*, 2022 ODVA Industry Conference), but it provides only minimal real-time metadata and offers no way to encode rich business logic. EDD improves on this with METHODS and calculated values, but its runtime integration story is limited outside of FDI. DTM/FDT packages are executable binaries with operating-system dependencies, which makes them expensive to maintain across platforms and impractical to run in modern container-based deployments. None of the three provides a unified, protocol-agnostic model for both device description and device behavior.

Many of these legacy systems assume a one-to-one service to user. In the cloud, having to maintain a service per user is very expensive, both in cloud infrastructure costs and requiring complex administration. The goal is to move device configuration to multi-tenant services. (Pulido, A., *Industrial Cloud Strategy*, LNS Research Research Spotlight, September 2026. [Industrial...earch 2026 | PDF]) However, any move to support multi-tenant can't then limit on-prem or mobile solutions as developing a solution for each is cost prohibitive.

The challenge is to break the descriptor complexity curve introduced in Section I. A purely data-driven descriptor represents increasingly complex devices by expanding their grammar, requiring every host to expand its interpreter in parallel. The solution is to let the descriptor define what a device *is* while moving what it *does* behind an API. Once behavior is exposed through a versioned contract rather than embedded in a static schema, three architectural questions arise: where state resides, how behavior is exchanged, and how the contract evolves. These questions lead to the three design commitments developed below.

B. Scope, Problem, Solution, Outcomes

Cloud applications adopt statelessness and multi-tenancy because those properties are what make scale economically possible. When a service holds no session state, any replica can answer any request: a load balancer can distribute work freely, new instances can be added under demand without draining connections, and failed instances can be replaced without losing in-flight context or corrupting the next request. Multi-tenancy extends that model one level up one deployment serves many customers or plants, with tenant identity carried in the request envelope and isolation enforced at the service boundary rather than by provisioning separate hardware per customer. Together, statelessness and multi-tenancy turn integration from a per-customer, per-host capital expense into a shared operational platform that grows with load. Device integration has historically resisted this pattern because DTM binaries and EDD engines pin state inside the host process; the architecture here deliberately relocates canonical state to the host service or to edge devices and keeps device logic stateless so the same scaling and isolation model that cloud software relies on applies to device configuration and control.

These concepts are not speculative. They were exercised in a working proof of concept (2024–2026) that delivered stateless TEMPO Services for two production device families. The core TEMPO Service collapsed to roughly 270 lines of code reusable across device types; end-to-end request cycles completed in well under 100 milliseconds; C#/.NET and Python components interoperated through CloudEvents without shared binaries; and automatic reconnection produced no observable state leakage under intermittent networks. The remainder of this paper develops the architecture behind those results.

C. A Proven Track Record for the Underlying Patterns

The 2024–2026 proof of concept is one concrete reference point for these claims. TEMPO Service logic collapsed to a small, testable core once persistent state and I/O were pushed out to the host and adapters; end-to-end request cycles ran well under 35 milliseconds when business logic was pure and state was event-sourced; the same TEMPO Service code ran unchanged behind a desktop client and a cloud host, confirming that the API contract not the runtime is the compatibility boundary; multi-language integration was a non-event, with C#/.NET and Python interoperating through CloudEvents; and intermittent connectivity was handled entirely by the transport layer, because the TEMPO Service held no state to leak.

The governance and convergence context for this work peer collaboration among ODVA, FCG, and other field-protocol organizations toward a protocol-agnostic integration technology, without

consolidating ownership of descriptors or certification is set out in Brooks, Hoeferlin, Wiberg & Vincent, *Device Integration Standards for EtherNet/IP*, 2025 ODVA Industry Conference. That paper describes how FCG and ODVA align on an **API-First** model in which independent services implement a shared, versioned contract while each body retains its own implementations and the existing installed base of EDS, EDD, DTM, and FDI packages remains supported. The approach used by **TEMPO** allows multiple existing standards to meet at the same API; rather than replacing EDS, EDDL or OPCUA it wraps and builds on it. The additional consideration is that vendors can provide custom solutions, similar to FDT but without the tight coupling to code APIs or particular operating systems. This would take a container first approach.

Every stakeholder in the current model pays a fragmentation tax, and the aggregate is large. Device vendors typically spend considerable time per device type to build and maintain the combination of DTM, EDD, UI, multi-host testing, and certification. Most EtherNet/IP devices, however, ship with only an EDS and none of these, so their integration cost is far lower; even then, producing a wholly new EDS still entails roughly a month of work once testing and release are included, while updates and derivative devices cost considerably less. Host vendors face have a very large spend per protocol for each of the protocol stacks, test harnesses, UI adapters, per device integration, and certification lab work. System integrators absorb a small to moderate amount per project in custom adapters, glue code, testing, on-site configuration, and operator training. End users are then impacted annually in tooling licenses, support contracts, engineer training, integration consulting, and downtime risk.

Stakeholder	Current Cost	Primary Pain Point
Device Vendor	Small to Medium	DTM (platform-specific) + EDD + UI + multi-host testing + certification
Host Vendor	Large	Protocol stack + test harness + UI adapters + per device integration + labs
System Integrator	Extra small to small	Custom adapters + glue code + testing + on-site configuration + training
End User	Extra small to Medium	Tooling licenses + support + training + consulting + downtime risk

FDI unifies process-automation integration around EDDL (IEC 61804): an EDD, optional User Interface Description and HTML5 plug-in, and a catalog entry, typically delivered through FDI Sync (Smitha & Chatrapathi, *EtherNet/IP FDI*, 2018 ODVA Industry Conference). Its strength is rich declarative business logic — METHODS, dependencies, and calculations — but the EDD engine is tightly coupled to the FDI host, and FDI offers little guidance on stateless, event-driven, or cloud-scale operation.

FDT integrates devices through executable **DTM** packages coordinated by an FDT frame application. DTMs offer maximum vendor flexibility and broad protocol support, including EtherNet/IP, but as opaque, platform-dependent binaries they are hard to audit in isolation and

 ©2026 ODVA, Inc.

poor candidates for cloud, container, and multi-tenant deployment. The practical path is preservation through a shared API wrapper rather than forklift replacement.

EDS is a structured, machine-readable device description that already carries identity, parameters, connections, and configuration metadata TEMPO can import directly (Wiberg Industry Conference; Majcher & Schaffner, *Metadata*, 2025 ODVA Industry Conference). For simple devices, the TEMPO Framework Library — a standard wrapper around an EDS that implements the TEMPO API — can produce a functional TEMPO Service from the EDS alone; for complex devices, the TEMPO Framework Library supplies structure and the vendor adds remaining behavior as pure functions. The same EDS continues to serve legacy hosts while the generated TEMPO Service serves TEMPO-enabled hosts, so adoption can be incremental without duplicating prior work.

TEMPO borrows FDI's business-logic expressiveness and FDT's protocol-agnostic reach, and adds explicit stateless responsibility boundaries plus CloudEvents-over-MQTT coordination. A shared ODVA/FCG Unified API could wrap EDS, FDI and FDT as implementation variants beneath a common contract during migration. Figure 1 shows how the core features of EDS, FDI, and FDT overlap. TEMPO Allows these to co-exist within the same application with a single network based API.

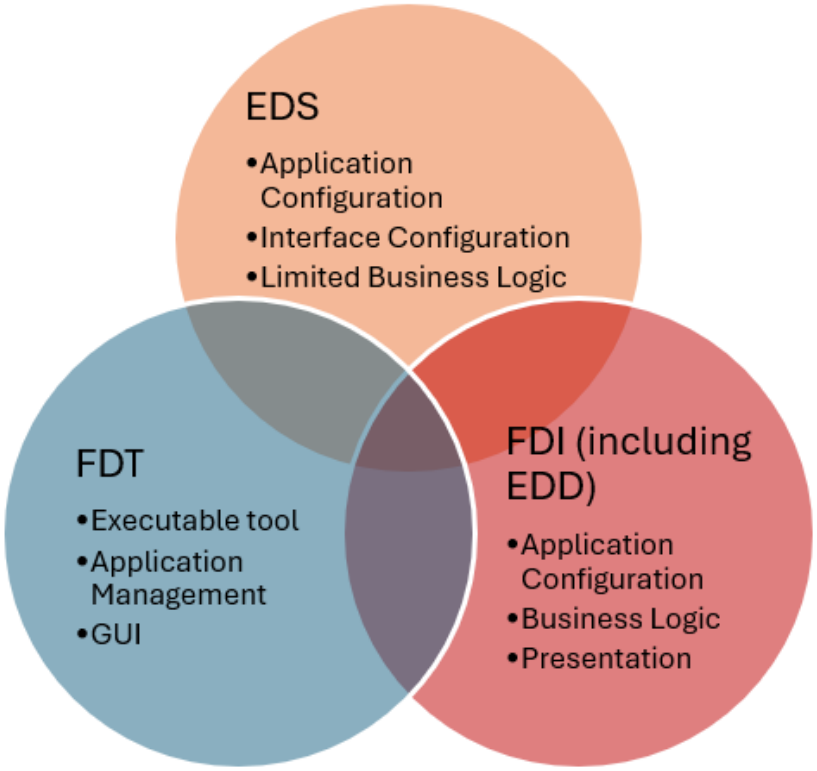


Figure 1 Core features of EDS, FDI, and FDT

IV. Comparison: TEMPO vs. EDS vs. FDI vs. FDT

The four technologies differ most in what they can express, where they keep state, and how far they reach toward cloud, edge, and multi-tenant operation. EDS is the baseline: a static, declarative data sheet that describes what a device *is* — identity, parameters, connections — with no business logic and no runtime state of its own, which is exactly why it composes so well as the

foundation the others build on. FDI adds rich business logic but keeps state in an EDD engine tightly coupled to the host; FDT adds maximum flexibility but keeps state inside an opaque binary DTM; TEMPO keeps canonical state in the host and makes the TEMPO Service stateless. EDS is EtherNet/IP-native and ODVA certified; FDI targets HART, FOUNDATION Fieldbus, and PROFIBUS PA; FDT is protocol agnostic but vendor-specific; TEMPO is protocol-agnostic and designed for cloud and edge. None of EDS, FDI, or FDT was designed for cloud scaling, multi-tenancy, asynchronous execution, or explicit concurrency control, whereas each is native to TEMPO. Crucially, TEMPO does not displace EDS — it consumes it (Section III.C), so the comparison below is less "which one wins" than "how each layer contributes," with EDS supplying the descriptor foundation beneath the richer behavioral models.

 ©2026 ODVA, Inc.

Dimension	EDS	FDI	FDT	TEMPO
Primary role	Static device descriptor (what a device*is*)	Descriptor + business logic	Executable device driver	Descriptor + stateless behavior (what a device *does*)
Business logic	None (declarative data only)	Rich (EDDL METHODS, calculations)	Arbitrary (binary code)	Rich (pure functions; composable)
State model	Stateless file (no runtime state)	EDD engine and DMS manages state	DTM owns state (binary)	Host owns canonical state; TEMPO Service stateless
Protocol support	EtherNet/IP (CIP)	HART, FF PA, PROFIBUS PA	All (vendor specific)	All (protocol agnostic; ODVA certified)
UI flexibility	Parameter groups only (no UI logic)	UID + optional HTML5 UIP	HTML5 (recent DTMs)	View models + custom HTML5
Cloud / scaling	N/A (static artifact)	Limited - (coupled engine, stateful)	Limited (OS dependent binary, stateful)	Native (stateless; MQTT-based)
Multi-tenant	Not applicable	Not addressed	Not addressed	Built-in (event partitioning)
Async execution	Not applicable	Not modeled	Not modeled	CloudEvents + adapter pattern
Concurrency control	Not applicable	Basic	Basic (vendor specific)	Optimistic (commit-version based)
Backward compatibility	The baseline (installed base)	N/A (new standard)	Via FCG adapter layer	Reuses EDD/EDS; wraps DTM (no forklift)
Event sourcing / audit	Not applicable	Not implemented	Not implemented	Full (host-managed immutable log;

 ©2026 ODVA, Inc.

				aids SOX/FDA/IEC 61508)
DevOps readiness	N/A (text file, version controllable)	Emerging	Limited	Full (containers, orchestration ready)
Role in TEMPO	Source artifact TEMPO imports/generates from	Wrapped via FCG Unified API	Wrapped via FCG Unified API	The unifying contract

A. An API-First Abstraction Over a Message Bus

The event-based API over a message bus is not merely a transport choice; it is the primary abstraction that lets TEMPO stay compatible with legacy systems while remaining open to future innovation. The governing principle is that any technology, host, or integration approach can participate if it can speak the API. The message bus decouples *who* is communicating from *how* they are implemented. This has three consequences. Existing DTM-based hosts, EDD engines, and legacy AOP profiles can interact with TEMPO through CloudEvents contracts by way of a host adapter that translates legacy call semantics into API messages, with no change to the device package. New approaches such as an AI-driven configuration agent, a mobile management client, a cloud SCADA platform, a serverless edge function, connect simply by producing and consuming the same CloudEvents, with no new protocol and no binary dependency. And the underlying transport can be upgraded or swapped entirely while every consumer keeps working, as long as the CloudEvents envelope is preserved.

The choice of a message bus over synchronous REST is deliberate. Device configuration is frequently long-running and multi-step; it must survive intermittent connectivity; it must fan out to multiple interested parties (a UI, an audit log, TEMPO Services, an analytics subscriber); and it must not assume an inbound network path to the device. Publish/subscribe messaging satisfies all of these naturally, whereas a synchronous request/response API fails the moment the network blinks or a customer refuses to open an inbound port.

B. A Stateless Device-Application Layer

The second commitment is that device-specific business logic runs as a stateless application. The TEMPO Service receives a request together with the current device models and host context, applies changes to device state, validates that state, and returns updated models, view models, and device state with any validation errors recorded in the state. The TEMPO Service does not simulate the device; rather, it embodies the device's business logic for example, if two options are mutually exclusive, it rejects a configuration that selects both. It holds no internal persistence and performs no side effects. State pinned inside a configuration process is what makes systems fragile, unscalable, and hard to reason about; a stateless service is interchangeable, so instances can be added or removed under load, updates can ship without draining sessions, and one failed request cannot poison the next. Statelessness also makes behavior deterministic and testable the same inputs always produce the same outputs which is the foundation for cloud scale, multi-tenant isolation, and honest audit trails.

C. Event-Driven Workflows

Complex operations multi-step calibration, cross-device coordination, deferred-apply configuration require synchronization that a single request/response cannot express. TEMPO handles these through explicit events on the message bus: the host coordinates the workflow by publishing events, the TEMPO Service stays stateless, and communication adapters (which may sit at the edge or inside a controller) subscribe and act asynchronously. This yields four properties that request/response cannot match. Execution is decoupled, so an adapter can run asynchronously, even at the edge, without blocking the host. The event log gives every state change a replay-friendly, immutable audit trail. Multi-tenant deployments partition event streams by tenant, providing isolation without shared state. And because every state change is observable as an explicit event, debugging and compliance auditing become tractable in ways they never were with binary DTMs.

Coordination across these asynchronous events is the host adapter's responsibility. It enforces a per-device, single-flight lane at most one state-changing operation in progress per device instance, with additional operations queued in arrival order so that a device instance behaves as a concurrency shard while operations across different instances proceed in parallel. This keeps the TEMPO Service stateless, because it never has to reason about interleaving, without sacrificing determinism. The precise token model and failure-and-recovery semantics are developed in Section IX.

D. Why the Three Commitments Reinforce One Another

These are not three separate bets. Statelessness is what makes the API-First contract scale, because any instance can serve any request; the message bus is what lets that contract reach the edge without special cases, because it is outbound-only, buffered, and reconnect-safe. Adopt one and the other two follow. Adopt all three and the modern usecase surface horizontal scaling, tenant isolation, edge relaying, mobile commissioning, multi-location editing, cross-tool import, AI-driven workflows falls out of the architecture instead of being bolted onto it. TEMPO is precisely the intersection of an API-First contract expressed as CloudEvents, a stateless TEMPO Service model in which the host owns canonical state, and a message-broker transport that lets edge agents, cloud services, desktop tools, and mobile clients all participate as equals.

V. Architecture Overview

A. A Four-Layer Responsibility Model

Responsibility Boundaries. TEMPO defines a strict division of responsibility so that state does not leak across architectural boundaries. Each layer owns a distinct concern: the client owns presentation, the host owns canonical state and coordination, the TEMPO Service owns stateless device-specific behavior, and the communication sub system which processes TEMPO communication requests into the OT communications to a physical device.

Device UI. The client owns the user interface. It renders view models device-specific UI descriptions with data attached sends mutations as CloudEvents, and tracks the commit version so a stale editor cannot silently overwrite a newer change.

Host. The host owns canonical configuration, which is the single source of truth for a device. For every incoming mutation, it checks preconditions such as concurrency and deduplication, invokes the stateless TEMPO Service, merges returned device state, advances the commit version, manages validation errors, and emits the response.

TEMPO Service. The TEMPO Service represents a specific device type within the TEMPO architecture. It is stateless: given a request, the current models, and host context, it validates the proposed device state and returns updated models and device state, with validation errors recorded in the state when constraints fail. It holds no persistence and performs no side effects.

Communication Sub System. A protocol-specific execution layer can apply accepted configuration to a physical device and report completion through CloudEvents. Where the underlying protocol requires connection or session state, this layer manages it explicitly, and its status can be surfaced back to the Device UI after it has been processed by the host which uses the TEMPO Service to interpret the responses to generate the information shown in the UI.

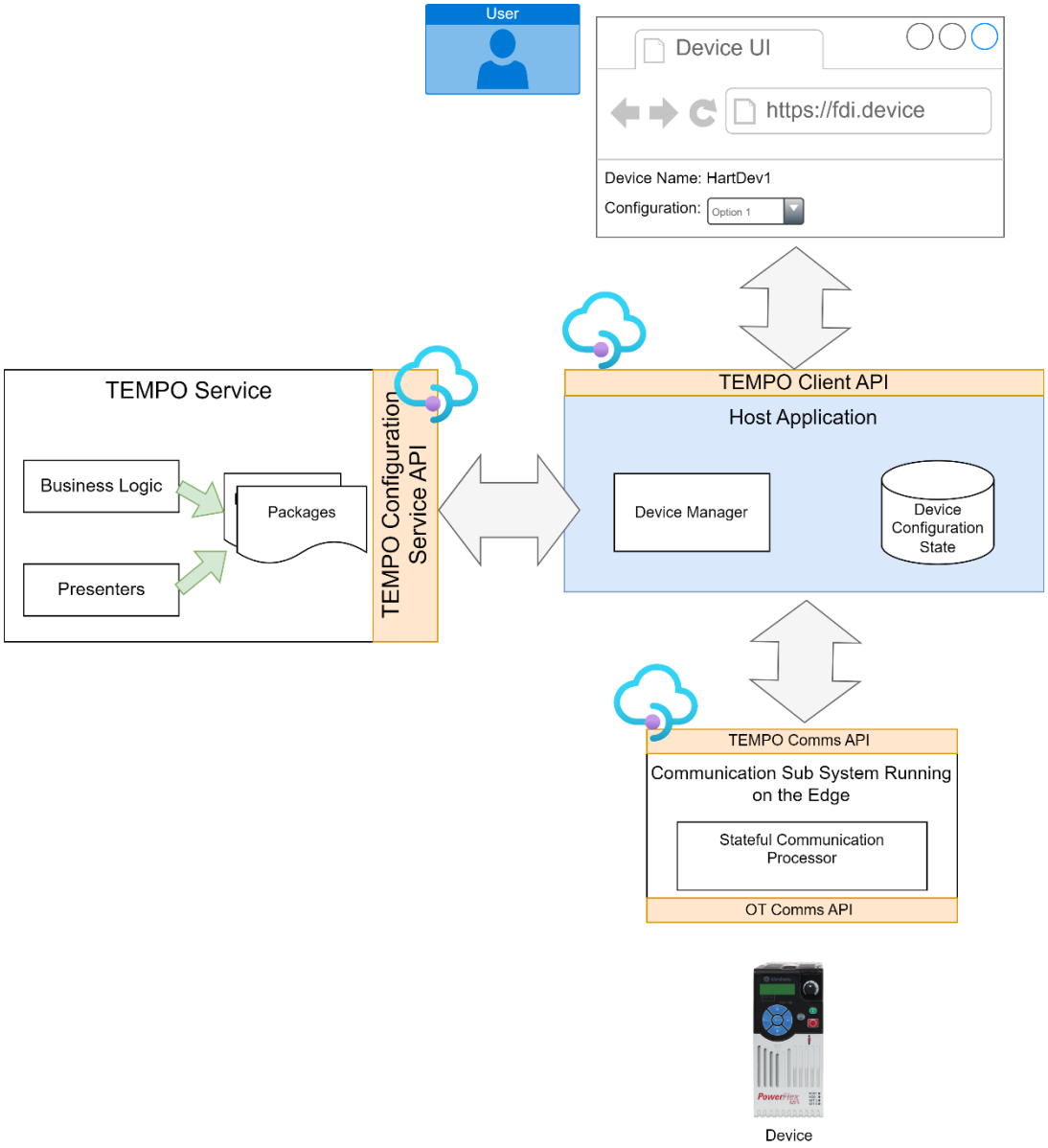


Figure 2 shows this host–device communication at a conceptual level.

B. Product Catalog

Several patterns recur throughout the architecture and, taken together, define what it means for an implementation to be "TEMPO-shaped." The host maintains a TEMPO Service registry indexed by device type, so EtherNet/IP, HART, and PROFINET devices can coexist and each request is routed to the correct TEMPO Service. Each service provides registration information about what products it supports. The host can then build up a catalog of supported products so as these products are added to the project or discovered on the network, the host application can then determine which TEMPO service supports configuring the device. The registration would also

 ©2026 ODVA, Inc.

provide information on what type of OT communication is required. Figure 3 shows the information model management components that supports the catalog and communications.

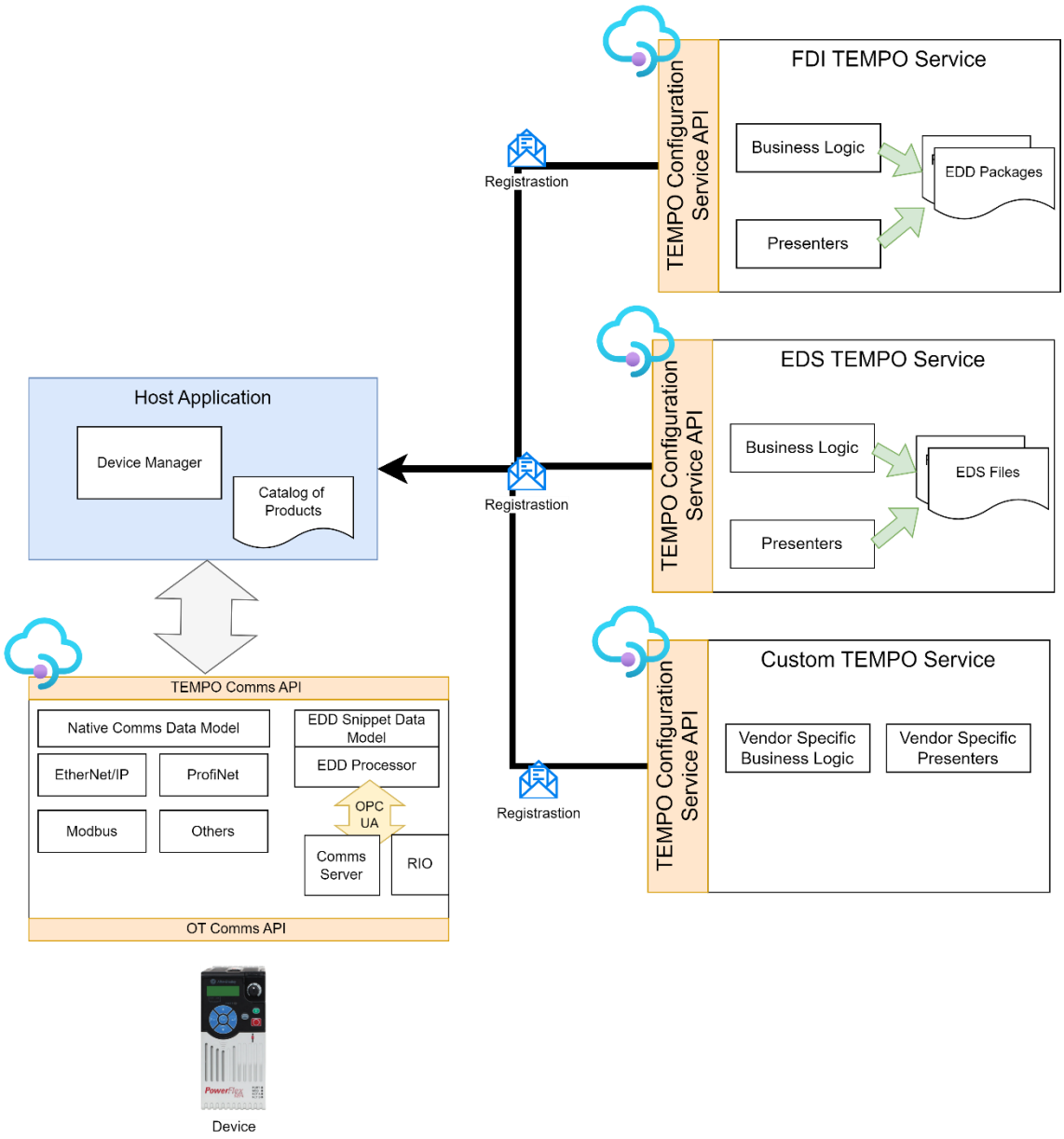


Figure 3 Information-model management components

C. The Canonical Device Model

Canonical State. The host-owned canonical state is not an opaque blob. It is a structured, protocol-agnostic device model that represents each device instance as an immutable record. Its top-level members carry identity, electronic keying, connections, data types, and configuration.

Identity and Keying. A device carries a CIP identity vendor ID, product type, product code, and major and minor revision that maps directly to the keying data used when a connection is established. The profile declares the keying policy, such as exact match, compatible module, disabled, or a custom key mask, along with the options offered to the user. The host then sets the actual key bits. This surfaces the same electronic-keying contract EtherNet/IP already enforces as

first-class model state, so a mismatch between a stored configuration and the physical device can be caught before a connection is opened.

Configuration Representation. Device configuration is modeled as a unified record that can include an optional tag reference, a raw value buffer with its dimension and data type, and an optional script. The same structure covers a user-visible typed configuration tag, hidden configuration members, a script that references a variable data buffer, or a script-only device. Values are carried as hex-encoded buffers that correspond to the underlying configuration attributes, allowing the neutral model to round-trip to the wire without loss.

Collections and Change Tracking. Composite members such as connections, ports, and data types are held as keyed collections rather than positional arrays. This allows a change delta to target a specific element unambiguously, even after the collection is reordered, and it supports the change-tracking design described in Section VIII.

Optimistic concurrency control is enforced by carrying the last-known commit version on every mutation and rejecting any mutation whose version is stale. Message processing is idempotent, because every event carries a correlation identifier and the host keeps a deduplication window, so the at-least-once delivery of a message never causes double execution. Execution coordination is asynchronous: the host emits an event to the adapter, the adapter acts and reports back, and no layer blocks on physical I/O. Canonical state is projected into an OPC UA information model in real time, so parameters appear as variables and business-logic constraints as methods. The TEMPO Service is a pure function, and canonical state is derived from an immutable event log rather than stored directly.

Reuse Across Devices. Modeling these elements explicitly lets a single stateless TEMPO Service serve many device families, while the host can reason about compatibility without embedding vendor-specific logic. Because the canonical state remains a faithful, queryable representation of the physical device and its history, it also provides a foundation for digital-twin scenarios that ODVA has explored previously.

VI. Use Cases

TEMPO has been designed against a catalog of more than twenty-five use cases spanning offline configuration, online configuration, cross-device integration, and cloud-scaling scenarios. Figure 4 shows the typical plant-wide system architecture these use cases target multiple controllers, drives, meters, and remote I/O on a unified network, with engineering and asset-management software above them. The offline and online workflows below are detailed in the working-group materials (API First Approach for FDI, Slides 9–13).

 ©2026 ODVA, Inc.

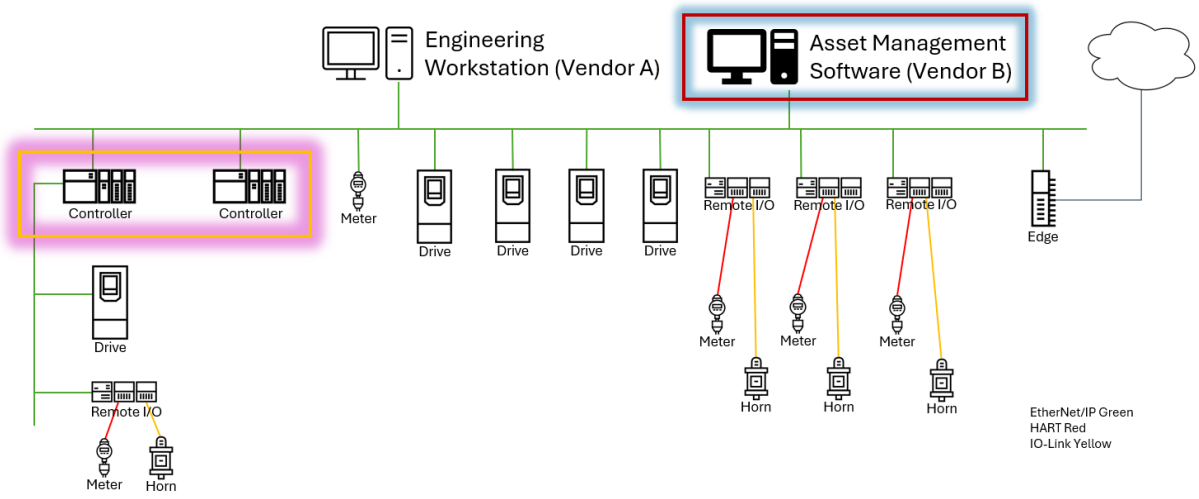


Figure 4 Typical plant-wide system architecture

In **offline configuration**, with no device connected, the host loads the device package and creates a canonical configuration seeded with defaults. Offline configuration is the model most commonly used in factory automation in which the controller and device configuration are both provided to the controller, with the controller then forwarding the device configuration at a later point in time. The user edits parameters through the offline UI; every edit sends a mutation carrying the current commit version; the TEMPO Service applies the change to device state, validates that state, and returns updated view models and device state with any validation errors recorded in the state; and the host advances the commit version and merges the returned device state into canonical state only when no validation errors are present. Export produces either a native configuration artifact the set of values needed to configure the EDS-defined device, expressed in the form the host application expects or a neutral JSON representation. This offline configuration can then be deployed to a controller which will then use the offline configuration to configure the device as part of its connection process. The host application would need to understand how to take the TEMPO device model and translate this into the model needed by the controller.

In **online configuration**, with a live device, the host connects through the appropriate adapter and queries the device for online state I/O values, diagnostics, and any device side dynamic constraints. When the user edits, the TEMPO Service validates the proposed device state against both the stored configuration and live constraints (for instance, the current operating mode may forbid certain changes), records any validation errors in the returned state, and returns that state to the host. Accepted changes state with no validation errors are emitted to the adapter, which applies them and reports completion; the host reflects the result into canonical state and notifies the client.

Cross-device integration treats the host as an orchestrator. A single client-initiated mutation may affect several device instances propagating a shared calibration factor across a sensor array or updating a controller and its attached I/O modules together and the host advances the commit version across all affected devices atomically in a single logical step. Coordinated, network-wide configuration of this kind is a direction ODVA has examined at the infrastructure level as well (*A Central Network Controller for Industrial Automation*, 2023 ODVA Industry Conference).

Cloud scaling and multi-tenancy fall out of statelessness. Because TEMPO Services hold no state, they deploy as ordinary stateless microservices replica sets, serverless functions, or on-demand containers. Event streams are keyed by tenant, subscriptions are filtered per tenant, and nothing crosses the boundary. Edge adapters subscribe to tenant-specific execution topics, and

the cloud-to-edge bridge is transparent at the CloudEvents layer, so an edge deployment requires no change to the TEMPO Service or the host's business logic. As plant networks add high-bandwidth wireless and 5G links between the edge and the cloud, this transport-agnostic split becomes more valuable rather than less (Jegajith & Chatrapathi, *Leveraging 5G Networks and EtherNet/IP*, 2023 ODVA Industry Conference).

VII. Stateless Business Logic Framework

A. Pure Device-Application Functions

The TEMPO Service is a functional API: given a request, the current models, and host context, it applies changes to device state, validates that state, and returns updated models, view models, and device state with any validation errors recorded in the state, with no I/O, no global state, and no side effects. The TEMPO Service is the deployment vehicle for this logic packaged as a container or an executable depending on the target platform. Because the same inputs the request together with the current configuration and device state supplied by the host always produce the same outputs, the TEMPO Service is deterministic, straightforward to unit-test, and safe to run as multiple concurrent instances against the same input without any synchronization. This is what allows the host, not the TEMPO Service, to own persistence and coordination — and it is why the same TEMPO Service code ran unchanged behind both a desktop client and a cloud host in the proof of concept.

B. Event Sourcing and Change Tracking in the Host

The host holds an authoritative event log, and canonical state is derived by replaying that log rather than stored directly. A device's state evolves through a sequence of events, each advancing the commit version: a parameter change, a validation pass, a submission, an execution completion. Nothing is deleted; snapshots exist only as an optimization; a temporal query becomes a replay up to a point in time; and disaster recovery becomes a replay forward from the last snapshot. Because each event application is a pure fold over the log, the same events always reconstruct the same state.

The same pattern powers change tracking and the user-visible "dirty" indicators in the UI. Rather than storing only the current model, the host keeps the original device state plus an ordered series of deltas, and the current state is the aggregation of those deltas. Each delta is expressed as a JSON Patch (RFC 6902) mandated by the TEMPO contract, not left to individual vendors, so hosts and TEMPO Services exchange changes portably a widely supported standard for describing the difference between two JSON documents which is exactly why the canonical model avoids positional arrays in favor of keyed collections, since a patch targeting an array index becomes invalid the moment the array is reordered. Every mutation returns the delta it produced, and the host appends it to the change list so that all services can reconstruct identical state from the same starting point. Each tracked change carries a small envelope of metadata: a strictly increasing commit version owned by the host adapter (the TEMPO Service treats it as pass-through), a context path identifying the target, the patch itself, and flags such as *redefined* (the change replaces the whole model at a path and serves as an aggregation point), *aggregated* (a periodic checkpoint that folds prior deltas into a snapshot), *reconfigure* (the change requires the controller to reconnect), and *applied* (whether the change has been committed to the host project).

A defining property is that the TEMPO Service is never aware of how the host commits changes. Under immediate commit, the host flushes each change to the project as soon as the mutation response returns, so the pending list is always empty. Under apply semantics, the host retains pending deltas until the user presses Apply or OK, treating the device model as a temporary entity

distinct from the rest of the project which is what lets a web client close and reopen without losing in-progress edits, since the host not the client persists the pending deltas and the event log. The TEMPO Service emits the same deltas either way; only the host's handling differs, and that is what makes one stateless TEMPO Service portable across a desktop tool with apply-and-cancel dialogs and a headless cloud workflow with immediate commit. Undo and redo then reduce to moving the head pointer within the delta series, so the undo stack is integrated into the architecture, and external changes a value edited in a tag browser, a version-control pull, an edit from another workstation enter the same list and are reconciled against pending user edits so the user can choose which to keep. Direct-to-device online values are the deliberate exception: they are applied immediately and are not part of undo, because a physical write to a device cannot be silently rolled back.

C. Canonical Device State and Validation

TEMPO Services express changes as updated device state. Validation is performed on that device state by the TEMPO Service a pure function that evaluates schema, range, dependency, and business-logic constraints against the proposed state and **records the results in the state itself**: when validation fails, the returned device state carries the validation errors; when it passes, those error fields are cleared. The host processes each response through a fixed pipeline: receive and unmarshal, inspect the returned device state for validation errors, merge state into canonical form when errors are absent, append the resulting event to the log idempotently, project into view models and any OPC UA models, and emit downstream events to clients, adapters, and subscribers. Because validation and state production are pure functions, the pipeline is deterministic, every outcome is auditable, and updates applied on multiple replicas converge on the same state.

A crucial invariant is that a response carrying validation errors, the errors are observable in the returned state, in the view models projected from it, and in the event log, but the device's committed configuration is unchanged until validation passes. This keeps the audit trail honest: a rejected edit is visible in history without having altered the device's stored configuration.

D. Protocol-Agnostic Schemas and Their Evolution

The canonical model describes parameter metadata (identifier, type, range, units, description), dependencies between parameters, business logic expressed as pure predicates, and versioning information. The model and message structure are described with a JSON Schema. The name have the JSON Schema is used to identify the information in the CloudEvent. There are 2 levels of versioning. The message structure and the data models are contained in the message. This enables granular control to limit the overall impact of a small change to a data model from impacting the overall implementation.

TEMPO Services evolve as parameters are added, ranges are adjusted, and pages are restructured. Without an explicit versioning strategy, a host could not safely deploy an updated TEMPO Service against configurations created by an earlier version of the TEMPO standard. TEMPO therefore carries schema-version metadata in every CloudEvent, uses semantic versioning for TEMPO Services, and recognizes three categories of change. Additive changes introduce new parameters with defaults, so old configurations keep working unchanged. Range adjustments preserve existing values or emit a warning when a value now falls out of range. Breaking changes like renamed or removed parameters require an explicit migration tool and cannot be auto-upgraded without user confirmation. The canonical state records which schema version produced it and which TEMPO Service version last modified it, and every migration is recorded explicitly so the audit trail shows when and why a device was migrated. When a client

 ©2026 ODVA, Inc.

connects, it advertises its preferred version; the host selects that version if available or offers alternatives; and every event in the session then carries the negotiated version.

VIII. Transport & Messaging

A. CloudEvents as the Canonical Envelope

Every message in TEMPO, a mutation, a response, an execution command, a completion, a downstream notification is a CloudEvent. CloudEvents is a CNCF (Cloud Native Computing Foundation) standard envelope that identifies the event, its source, its type, and its data, and it is deliberately transport-neutral. Standardizing on it means the payload identity survives regardless of which broker carries it, and it gives TEMPO official protocol bindings and SDKs across the common languages and transports without bespoke work.

B. IT Protocol Options

TEMPO is agnostic to the network transport in the cloud but there are some practical limitations. The TEMPO service will only support one or two transports otherwise the cost for creating and testing the service can grow without providing value. For example, HTTP2/REST/SSE and MQTT. However, the host is free to route these messages over other protocols. However, with the use of CloudEvents, host application developers can choose to route messages over buses or protocols of their choice using off the shelf gateways.

MQTT is the natural baseline: lightweight, open, widely deployed in industrial environments, and available in both self-hosted and managed forms. Its publish/subscribe semantics fit a multi-tenant, event-driven architecture, and its resource profile fits constrained edge hardware. Conceptually, four topic roles suffice client-to host mutations, host-to-client responses, host-to-adapter execution commands, and adapter-to-host completions. Kafka and Azure Service Bus are not MQTT brokers, but because the CloudEvents envelope and the CRUD-style message contract are the interface, the same events map cleanly onto Kafka partitions or Service Bus queues, so a host vendor can run TEMPO over messaging infrastructure it already operates.

A key design concern is how to route messages across many services and gateways without creating endpoint-by-endpoint complexity. If every service exposes its own HTTP or REST endpoint, the platform must manage routing, authentication, certificate distribution, and policy enforcement separately for each one. That burden increases when TEMPO is deployed across multiple cloud providers, user PCs, mobile devices, internet-connected plants, air-gapped environments, and combinations of those deployment models.

The topic convention is concrete and directional. Frontend-to-backend traffic is published under a `devices/tempo/{instance-id}/host/{api}` topic and backend-to-frontend traffic under `devices/tempo/{instance-id}/client/{api}` , with the API segment version-pinned. There is exactly one destination per device — no topics are permitted after the instance identifier which lets a gateway filter subscriptions by device so a wildcard cannot be used to observe unrelated traffic. Client operations ride as distinct CloudEvent types: mutate and query from the frontend; apply, ok, and close for commit and lifecycle control; and update from the host back to the frontend. Notably, failures do not get a separate type validation errors recorded in device state, concurrency conflicts, and other recoverable errors ride inside the update payload so a client inspects one event type for both success and failure. This single-destination, versioned-API discipline is what keeps the same client API usable across a single-user desktop bus and a multi-tenant cloud bus without topic collisions. Delivery uses at-least-once semantics, which, combined with deduplication, guarantees exactly-once *effect* even when the transport delivers duplicates.

C. Correlation, Concurrency, and Recovery

Three header fields coordinate the full lifecycle of a change, and they must not be conflated. An execution-attempt identifier deduplicates: it is generated per attempt, stays stable under a transport duplicate, and changes on every retry, so the host can drop duplicates and correlate a response to its in-flight attempt. A commit version enforces optimistic concurrency: it is a monotonically increasing integer owned by the host, advanced exactly once per accepted state change, and carried by a command as a precondition so a stale command is rejected before it does anything. A causation identifier provides lifecycle correlation: it is generated once on the initial command and propagated through every message the action triggers, staying stable across retries so an operator can trace a single logical action end to end.

The host adapter is the single coordination point, and a few invariants make the token model safe under real-world delivery. It runs a per-device, single-flight FIFO lane, so at most one state-changing operation is in progress per device instance while different instances execute in parallel. It rejects staleness before any side effect. It deduplicates on the attempt identifier. And it accepts exactly one externally visible state change per commit version, first-in wins; read-only queries bypass the lane, since they neither advance the commit nor produce side effects. A rejected command is not so much an error as a coordination signal, handled through an invalidate-and-replay exchange: the adapter emits an ordinary update carrying an invalidation payload — the reason, the rejected attempt, and the true current commit — while preserving the causation identifier, and the client refreshes and decides whether to recompute, resubmit under a new attempt identifier, or abandon. The infrastructure never retries automatically. This same mechanism absorbs the harder failures. A lost response is retried as a new attempt and any late original is dropped; an external host change invalidates outstanding client mutations; and an adapter crash simply drops in-flight work, because the current commit is re-read from the host on restart rather than reconstructed from volatile memory. All events triggered by one user action share a causation identifier, which makes multi-step workflows replayable and makes audit queries such as "show every change in this session" straightforward.

D. Reaching External Clients Through Gateways

MQTT is efficient as the internal backbone, but external clients live in a wider protocol universe browsers prefer Fetch and Server-Sent Events, mobile apps prefer WebSocket or gRPC, and legacy SCADA prefers HTTP polling. TEMPO keeps MQTT internal and exposes external protocols through purpose-built gateways that translate to and from the internal topics while preserving the CloudEvents envelope end to end. A REST gateway turns an HTTP request into a mutation event and waits for the response; a Server-Sent Events gateway subscribes on the client's behalf and streams events to a dashboard; a WebSocket gateway multiplexes mutations, responses, and events over one bidirectional connection for mobile and collaborative UIs; and gRPC and GraphQL gateways serve high-performance services and flexible data selection respectively. Legacy environments are reached the same way, with OPC DA and Modbus bridges mapping their operations onto the bus. In every case the envelope is preserved, so tracing and identity survive the crossing.

E. The Broker Is an Infrastructure Choice

Because CloudEvents is transport-neutral, the broker becomes an infrastructure decision rather than an architectural one. MQTT is the baseline and the right choice at the edge and on constrained hardware; its only notable limitation is that it retains a last message per topic rather than a replayable log, which does not matter because TEMPO's event log lives at the application layer regardless of transport. Host application vendors can also consider Apache Kafka fits cloud-scale deployments with high throughput and native offset-based replay, at the cost of heavier

operations unsuitable for the edge. Or Azure Service Bus and its equivalents fit enterprise integration with rich messaging semantics, at the cost of a hard cloud dependency. A hybrid MQTT at the edge bridged to Kafka in the cloud is a common industrial pattern, and the CloudEvents envelope is what lets an event keep its identity as it crosses the bridge. The TEMPO Services, host logic, and event-sourcing patterns are unchanged by the choice.

F. A Data-Driven API That AI Can Use

At its core the TEMPO API exposes a data-driven-style (CRUDish) contract for device configuration create, query, and update expressed as CloudEvents, and every payload carries not just raw values but the schema metadata (type, range, units, dependencies) needed to reason about validity.

Because each parameter either carries its constraints or can load them statelessly, a user or AI assistant can answer questions such as "what is the valid range for this calibration offset?" without embedding device-specific logic. TEMPO Services make these interactions predictable through explicit functional contracts. Validation runs against device state and records field-level errors there, so clients and assistants can inspect diagnostics directly and refine proposals iteratively. Deterministic results reduce a major source of AI hallucination, while the immutable, queryable event log allows an assistant to explain why a parameter changed, who changed it, and when. This aligns naturally with emerging agent frameworks such as the Model Context Protocol, where data-driven operations become tools and device models become resources a much different model than calling a black-box binary and receiving an opaque error code.

IX. Integration with Standards

TEMPO's integration strategy is reuse and gradual migration, not replacement. For FDI, existing EDD, UID, and UIP files are repurposed as metadata sources: the host reads them and maps their business logic onto the canonical device state model, the FDI catalog is retained, and where an FDI Sync API is available the host queries it for package metadata. All existing FDI packages remain usable with no re-export, and an import tool automates the EDD-to-SPI migration while preserving semantics. The FCG Unified API then abstracts over both FDI and FDT so either legacy path can serve a TEMPO host: DTM binaries keep running in containers or sandboxes with their calls translated to TEMPO contracts, EDDengine outputs are routed through the same API, and vendors migrate to native TEMPO Services incrementally with no forklift replacement. Automated converters preserve parameters and business logic where possible and flag complex logic for manual review, and version management records the lineage of every configuration so a migration can be rolled back if it introduces a defect.

For EtherNet/IP specifically, EDS is the entry point: its metadata converts to a parameter schema, its real-time I/O tags become observable state in view models, and its vendor extensions become custom properties. The preservation guarantee is strong existing EDS files are never deleted or modified, they remain authoritative for legacy hosts, and a vendor may maintain both an EDS and a TEMPO Service simultaneously, migrating at its own pace or keeping EDS alive indefinitely. The EDS TEMPO Service when containerized contains both the business logic to configure, present and communicate with the device identified by the EDS file but also contain the EDS files themselves. By putting these into a container we can take advantage of the container technology which ensures the container has not been modified. Because the EDS remains a security-relevant artifact, any import path into the host application must respect the ways an EDS can be compromised that ODVA has documented (Visoky & Wiberg, *EDS Threats and Mitigations*, 2025 ODVA Industry Conference) validating and sandboxing imported files rather than trusting them implicitly. Distributing EDS files with the container the threat surface is reduced. For the case

where the user needs to provide a EDS file to the EDS TEMPO container for a device which was not included, any thing a compromised EDS could do would be limited to that container and not disrupt the system.

TEMPO and OPC UA are complementary, not competing. Canonical state maps naturally to an OPC UA information model under IEC 62541: parameters become variables, business-logic constraints become methods, and the commit version is exposed through the browse name so external clients can inspect historical states. This projection follows ODVA's established CIP-to-OPC UA mapping principles (Majcher, *CIP to OPC UA*, 2022 ODVA Industry Conference). OPC UA method calls translate into mutation events, variable reads translate into queries, and alarms and conditions map to completion status. As a result, an OPC UA client can drive TEMPO configuration through native OPC UA semantics without speaking CloudEvents directly. TEMPO owns the stateless configuration and communication semantics; OPC UA remains the runtime information model.

X. Realizing the Architecture: Approaches and Considerations

This section surveys how TEMPO could be realized in practice. It is not a committed roadmap; it is a discussion of the options, trade-offs, and open questions any implementation effort would face. The 2024–2026 proof of concept already provides concrete evidence of feasibility: roughly 270 lines of core TEMPO Service code reused across two product families, end-to-end cycles under 100 milliseconds, a C#/.NET core with Python activities, and automatic reconnection with zero state leakage — validating MQTT, CloudEvents, and the layered architecture together.

A. Cost Reduction: Mechanisms and Trade-offs

Several levers could reduce the fragmentation cost, each with a genuine trade-off. A standardized SPI a small, fixed set of operations every TEMPO Service satisfies creates a single test harness and predictable host integration, but a rigid SPI can limit unusual workflows and itself needs a versioning strategy. Open-source reference artifacts and SDKs could compress months of per-vendor work into weeks, but only if they are sustainably maintained; a stale reference implementation is a liability. Automated certification tooling can catch the large majority of structural issues before human review, but cannot judge whether a TEMPO Service behaves correctly for its physical device, so a hybrid of automated structural checks and human behavioral review is required; as TEMPO is built on top of the existing infrastructure it can leverage the conformance-testing infrastructure ODVA has developed over more than a decade (*Developments in Conformance Testing*, 2015; *Network Conformance Testing*, 2017; *Conformance Testing*, 2022; *Conformance Authority*, 2025). The API and functional nature simplifies the testing as it reduces the possible interactions. Stateful services are noted to be difficult to test because many tests require pre-conditions. The use of HTML for the user interface allows testing using off the shelf test tools to validate the correct information is being display for any particular condition.

Cloud-hosted catalog services could cut per-seat licensing substantially while improving discoverability, but introduce availability and data-residency concerns that demand an offline mode for regulated or air-gapped plants by centrally running the services from servers within the plants. The FCG Unified API can act as a migration shim so TEMPO hosts coexist with FDI and FDT, but wrapper layers add latency and their fidelity is bounded by how well a DTM can be instrumented some DTM behaviors may not be wrappable at all. And mapping canonical state onto OPC UA lets existing OPC UA investments participate without new tooling, at the cost of careful design around versioning and concurrency to avoid an impedance mismatch. Indicatively, and assuming broad adoption, these mechanisms could change the time to develop from 6 to 12

 ©2026 ODVA, Inc.

months to 1 or 2 months, host-vendors would see similar magnitude cost reductions, but the actual savings depend entirely on adoption and on how well the shared tooling is maintained.

B. Configuration File Preservation

The guiding principle for preserving existing investments is import-and-derive legacy files are treated as source artifacts from which a TEMPO Service is *derived*, with the original remaining authoritative and the derived TEMPO Service regenerable if the source changes. This avoids the forklift-replacement problem while providing a path forward. EDS files are well-structured text, so parameter extraction is largely mechanizable; the open question is what fraction of real-world files carry implicit interdependencies that require manual review, which a sampling study across member EDS files would need to quantify. EDD carries richer business logic, and translating its constructs into stateless pure functions is conceptually straightforward but raises a real question about inherently stateful behaviors such as calibration wizards, which must be modeled either as explicit request types or by externalizing wizard state to the host. Binary DTMs are the hardest target: behavioral instrumentation in a sandbox is largely automatable for well-behaved DTMs but degrades sharply for those relying on COM or Windows internals, so the more practical path is to work with DTM authors on native implementations while retaining the DTM as a fallback. During any transition a device will likely support both a legacy path and a TEMPO path at once, and the certification framework will need to accommodate dual-format submissions and preserve the provenance chain which source file, which conversion tool, which approving engineer since that history is itself a compliance requirement in safety regulated environments.

C. Containerized, Sandboxed Packaging of Configuration Files

One of the most direct security benefits of the API-First model is that it changes *where* a device-description file is parsed. In the classic model, a host application reads an EDS, EDD, or DTM directly: it opens the file, runs it through its own interpreter, and executes whatever business logic the file implies all inside the host process, with the host's privileges and access to the host's filesystem, memory, and network. That is a broad attack surface. A device description is untrusted input, and the ecosystem has documented how a malformed or malicious EDS can be turned against a host (Visoky & Wiberg, *EDS Threats and Mitigations*, 2025 ODVA Industry Conference). Because the file and the host share a process, a parser bug becomes a host compromise.

The container model inverts this. Recall that each standards body and each device vendor packages its logic as its own container (Section II.C): the FCG can provide containers that manage FDI packages, ODVA containers can handle EDS packages, and a device vendor can ship its own container in whatever language it chooses. When a configuration file is consumed *inside* that container, the untrusted file never touches the host. Additionally, the container can itself include all or a selection of EDS files ensuring these files cannot be tampered with. The container parses the EDS or EDD, runs the device-specific logic against it, and exposes only the stateless API CloudEvents carrying canonical models, view models, and device state across the boundary. The host sees validated, well-typed data on a message bus; it never sees the raw file, never runs a device-supplied interpreter, and never grants a device description access to its own address space.

This yields a defense-in-depth posture that maps naturally onto the four API-First properties. Sandboxing is enforced by the container runtime: file parsing, script execution, and any legacy DTM binary run under a constrained profile restricted filesystem, no ambient network, dropped privileges, and resource limits so a hostile file is contained by the operating system, not merely by careful coding. Zero-trust is enforced at the message boundary: nothing crosses except signed, schema-validated CloudEvents, so a compromised device container cannot reach laterally into the

host or into another tenant's container. Blast-radius is bounded: if a malicious EDS crashes or subverts its parser, the damage is confined to that one container instance, which the orchestrator can kill and replace, rather than taking down the host that dozens of devices depend on. And provenance is preserved: because the container image itself is versioned and cryptographically signed through the catalog trust chain, an operator can attest exactly which parser and which device logic processed a given file. The practical upshot is that legacy artifacts including binary DTMs that cannot be trusted on a modern host can be run safely by isolating them inside a sandboxed container and letting only the API cross the wall, which is precisely what makes the "wrap, don't rewrite" migration path (Section IX) tenable without importing the legacy security surface along with the legacy behavior.

D. Alignment with the Margo Initiative for Edge Interoperability

The stateless, containerized, API-First shape of a TEMPO Service is not only good security hygiene; it is exactly the shape that emerging edge-orchestration standards expect. The **Margo** initiative an open project under Linux Foundation governance, defines mechanisms for interoperable orchestration, at scale, of edge applications, workloads, and devices, delivered as an open specification together with a reference implementation and a compliance-testing toolkit (margo.org). Margo's concern is the multi-vendor edge problem: how heterogeneous applications and devices from different suppliers are deployed, updated, and managed together without bespoke integration for every pairing. That is a complementary layer to TEMPO, not a competing one, and the two composed cleanly because TEMPO services as well as the communication sub system are by nature designed for containerization. Which Margo is designed to orchestrate.

The alignment follows the three commitments developed in Section V. Because a TEMPO Service is **stateless**, it behaves like a replaceable workload: an orchestrator can start, stop, scale, or move it across edge nodes without draining sessions or migrating in-process state. Because it is **API-First**, its external contract is CloudEvents on a message bus, letting Margo manage it as a black-box workload with a standard interface rather than understanding device-specific internals. Because it is **edge-native** outbound-only, offline tolerant, and packaged as a sandboxed container (Section XI.C) it can fit into a Margo-managed edge fabric without changing the business-logic layer. In a combined deployment, Margo defines how the workload is deployed, versioned, and kept healthy across edge nodes and vendors; TEMPO defines what the workload does and how it configures devices safely and deterministically. Concretely, a device vendor could publish a TEMPO Service container to a Margo-managed catalog, have a Margo-conformant orchestrator deploy it to plant edge nodes, and then use TEMPO's CloudEvents API to configure devices. In that model, ODVA owns the device-integration contract, Margo owns the edge-orchestration contract, and the sandboxing model limits implicit trust across the boundary.

E. Open Questions

Five questions dominate any realization. Governance: who owns the specification a joint ODVA/FCG working group, an independent foundation? Versioning: how are breaking SPI changes managed so a TEMPO Service written for one version keeps working with a host built for the next? Security: how are credentials managed for multi-tenant CloudEvents over MQTT, and how does that integrate with industrial identity systems? A question that should be answered within the holistic IIoT security posture ODVA has advocated (Arun & Gregory, *Holistic Cybersecurity for IIoT*, 2018 ODVA Industry Conference), and with the EDS-specific mitigations noted in Section X. Offline-first: the architecture supports offline operation by design, but the catalog, certification, and tooling all carry implicit connectivity assumptions that must be examined. And, most importantly, ecosystem momentum: the largest risk is not technical but adoption, because a sound standard with few implementations delivers no value.

XI. Ecosystem Benefits

For **device vendors**, TEMPO offers a single package that works with any TEMPO host across EtherNet/IP, HART, and PROFINET through the FCG adapter layer, replacing the current mix of DTM, EDD, and per-host testing. Stateless contracts make host behavior predictable, an SDK shortens development, and OPC UA exposure comes for free because the host projects canonical state into a namespace. The economics are favorable: where a DTM path cost $500K–$2M with OS lock-in and per-host certification, a stateless TEMPO Service built on open-source SDKs and an automated EDS mapping tool costs a fraction of that, break-even arrives after the first device, and subsequent devices cost roughly a tenth of the DTM path. Existing EDS, EDD, and DTM configurations remain valid and importable, with no forced updates, so a device can expose both a legacy EDS and a TEMPO Service during the transition.

For **host and integrator vendors**, the benefits follow directly from statelessness: simpler integration and testing when TEMPO Services have no hidden state, cloud deployment for free because TEMPO Services are ordinary microservices, multi-tenant SaaS because partitioning is built into the envelope, and OPC UA available as the discovery and databinding layer. Where custom integration ran to millions per protocol, a TEMPO host implementation plus a small per-device SPI integration cost brings the amortized cost per device down dramatically, with break-even after roughly ten certified devices.

For **end users and plant engineers**, the benefits are practical rather than architectural: the same tools across discrete, hybrid, and process automation; devices from different vendors working together under one certification; standards-based tooling that removes lock-in; deterministic workflows with audit trails and concurrency control; and coexisting online and offline workflows. The immutable event log managed by the host application meets SOX, FDA, and IEC 61508 requirements, which reduces compliance overhead, and IT teams gain ownership of configuration rather than depending on outside consultants for every change.

For **standards organizations**, TEMPO lowers the cost of lab operations automated SPI validation, standardized harnesses reused across vendors, and continuous regression suites while raising throughput, so more vendors are certified per year. It also creates a new strategic role: a cloud-based, jointly governed device catalog service (FDI Sync– compatible) could serve certified EDS metadata, TEMPO Service packages, and OPC UA models over an API with automated verification, cryptographic signing under an ODVA/FCG/OPC Foundation trust chain, real-time update propagation, and safe rollback. Such a catalog makes the standards bodies the authoritative source of truth for device integration, and its network effects grow the ecosystem as the catalog grows.

XII. Conclusion & Call to Action

For devices and solutions that are not restricted by the use of EDS today, TEMPO must, can and shall ensure that vendors can continue to provide those EDS files with no (or negligible) incremental work

TEMPO addresses a well-documented, costly problem in industrial device integration, and it does so with a technically coherent solution grounded in established patterns: stateless services, event sourcing, functional device logic, and CloudEvents. Whether and how it is realized depends less on technology than on governance, ecosystem momentum, and the willingness of ODVA, FCG, and their members to invest in shared infrastructure. The technical risk is manageable; the governance and adoption risk is substantial.

 ©2026 ODVA, Inc.

Honest advocacy should name the exposed edges. Ecosystem maturity is the largest: FDI and EDS have decades of vendor investment, certification programs, and installed tooling, whereas TEMPO has a specification, a proof of concept, and internal adoption but no third-party device vendors, no independent host implementations, and no conformance authority equivalent to FDI-package certification. Governance is still in flight, the FCG/ODVA alignment that would make TEMPO a cross-vendor standard is a work item, not a settled outcome so until it lands TEMPO looks, from outside, like a vendor specific architecture. Statelessness relocates state rather than eliminating it: the broker, the configuration store, and the event log now carry the availability, throughput, and security burden once spread across host processes, and a misconfigured access-control list or a leaky tenant claim can breach isolation in ways a monolithic host could not. Realtime and safety scenarios are not yet demonstrated at the level FDI and EDS support today, and hard-real-time control loops remain explicitly out of scope for the message broker path. And TEMPO does not delete the legacy surface DTM binaries, .NET UIPs, and Windows-bound services still exist and must be wrapped so the near-term reality is a hybrid deployment with more moving parts, not fewer, until adoption catches up with the architecture.

Positioned against those risks, the strategic alignment is nonetheless clear. For ODVA, TEMPO could become the EtherNet/IP device integration path, complementing EDS for new devices while preserving existing investment. For FCG, it could unify FDI and FDT under a single SPI so one device package works across host platforms and protocols. For the OPC Foundation, TEMPO's information models map naturally onto OPC UA, offering standards-based runtime interoperability without a new modeling language. If the standardized SPI, open-source tooling, and cloud catalog services described here can be realized and adopted, the cost and complexity of device integration across EtherNet/IP, HART, and PROFINET could fall dramatically not because the architecture is novel, but because it applies proven cloud-native patterns to a domain that has resisted modernization.

Several questions are worth putting to the community directly: whether TEMPO should be a joint ODVA/FCG standard, an open-source project, or a new working group; whether its scope should stop at configuration or extend to runtime and execution semantics; what level of EDS, EDD, and DTM compatibility the market actually requires; what would motivate the first device vendors to invest before the host ecosystem matures; and whether OPC UA should be treated as an optional projection target or as the primary information model. Concrete next steps follow from these: present the architecture to the ODVA, FCG, and OPC Foundation working groups; build a follow-on proof of concept focused on ecosystem breadth one device, one host, one EDS import to test the architecture against real-world EDS complexity; draft a core specification if there is sufficient interest; and survey vendors on current integration costs and their willingness to invest in a shared alternative. The vision that motivates the work is a single integration standard spanning factory, hybrid, and process automation, many protocols served by unified tooling and a single certification path, and a cloud-ready foundation that scales from the edge to the enterprise.

Appendix: References & Further Reading

A. Standards Documents

- FCG FieldComm Group — FDI Specification Suite
- ODVA EtherNet/IP Specification
- IEC 61804 (EDDL) Standard
- CloudEvents v1.0.2 Specification
- MQTT v3.1.1 Protocol
- IEC 62541 (OPC UA)
- Margo — Edge interoperability for industrial automation ecosystems (Linux Foundation)

 ©2026 ODVA, Inc.

B. Prior ODVA Conference Papers (Sources Cited)

This paper builds on prior ODVA Industry Conference work; the following are cited in the text as supporting material.

Brooks, Hoeferlin, Wiberg & Vincent — Device Integration Standards for EtherNet/IP (2025) — API-First architecture, FCG/ODVA convergence, and the figures used throughout this paper

Visoky & Wiberg — EDS Threats and Mitigations (2025) — security considerations for importing EDS files

Majcher & Schaffner — Metadata (2025) — device metadata enrichment

Voegel et al. ODVA Process Device Profiles Come to Life (2023)

Didier EtherNet/IP in process automation A Central Network Controller for Industrial Automation (2023) — network-wide coordinated configuration

Jegajith & Chatrapathi — Leveraging 5G Networks and EtherNet/IP (2023) — edge-tocloud connectivity

Majcher — CIP to OPC UA (2022) — CIP-to-OPC UA information-model mapping

Frazer & Snide — Current Developments in xDS Device Descriptions (2022) — EDS/xDS evolution

Choudhry & Drackley Conformance Testing (2022) — conformance-testing developments

Smitha & Chatrapathi — EtherNet/IP FDI (2018) — FDI on EtherNet/IP

Snide & Harriman — Digital Twin in Automation (2018) — digital-twin representations

Arun & Gregory — Holistic Cybersecurity for IIoT (2018) — IIoT security posture

ODVA A Day in the Life of Field Device Users (2017) — field-device user experience Network Conformance Testing (2017) and CIP Safety Conformance Testing (2017)

Blair — A Modern Approach to CIP Device Descriptions (2015),

Wiberg & Schiffer Optimizing EDSs for EtherNet/IP I/O Connection Target Devices (2015), and Developments in Conformance Testing (2015)

Pulido, A., *Industrial Cloud Strategy*, LNS Research Research Spotlight, September 2026. [Industrial...earch 2026 | PDF]

****

The ideas, opinions, and recommendations expressed herein are intended to describe concepts of the author(s) for the possible use of ODVA technologies and do not reflect the ideas, opinions, and recommendation of ODVA per se. Because ODVA technologies may be applied in many diverse situations and in conjunction with products and systems from multiple vendors, the reader and those responsible for specifying ODVA networks must determine for themselves the suitability and the suitability of ideas, opinions, and recommendations expressed herein for intended use. Copyright ©2026 ODVA, Inc. All rights reserved. For permission to reproduce excerpts of this material, with appropriate attribution to the author(s),

please contact ODVA on: TEL +1 734-975-8840 FAX +1 734-922-0027 EMAIL odva@odva.org WEB www.odva.org. CIP, Common Industrial Protocol, CIP Energy, CIP Motion, CIP Safety, CIP Sync, CIP Security, CompoNet, ControlNet, DeviceNet, and EtherNet/IP are trademarks of ODVA, Inc. All other trademarks are property of their respective owners.