## **Executive Summary — OPS MES**

## **OPS** is a modular Manufacturing Execution System that connects shop-floor events to business outcomes. It orchestrates production, quality, and logistics by integrating plant data (sensors/PLCs, historian) with enterprise systems (ERP/LIMS) and line applications (e.g., Harford), delivering real-time visibility, traceability, and continuous improvement across sites.

## Original OPS Document

## **What OPS Delivers**

## **End-to-end integration:** Secure Web APIs link OPS with **ERP (SAP)**, **LIMS**, **Harford**, and **weighbridge** systems for bi-directional data exchange (orders, confirmations, test results, weights), ensuring a single source of truth from intake to dispatch.

## Original OPS Document

## **Operational control & insight:** Historian connectivity (primarily **AspenTech IP.21**) captures time-series tags; OPS turns them into batches, steps, and events for monitoring and analysis. An **OEE** model visualizes uptime/downtime and quantifies losses to target the biggest performance gains.

## Original OPS Document

## **Modern user experience:** A **HTML5 Web Client** (Angular/PrimeNG) runs in any browser alongside the legacy **WPF** client, enabling multi-platform access and zero-touch updates. The **ConfigTool** gives administrators a secure, role-based UI to manage users, reasons/states, attributes/units, and integration setup.

## Original OPS Document

## **Architecture at a Glance**

## **Databases:**

## **OPSDB (SQL Server):** live, transactional backbone for batches, movements, users/roles, states/reasons, audits, and reporting SPs/views.

## **Integration Config DB:** isolated catalog for connectors, mappings, and schedules—validate in Dev/Test and promote to Prod without touching operational data.

## **OPS Data Warehouse:** star schema (facts/dimensions) for fast BI and deep-dive analytics (e.g., max-granularity OEE).

## Original OPS Document

## **Services:**

## **Integration Service:** multi-instance (by *ServiceID*), translates historian/OPC/file inputs into structured OPS events (e.g., CIP batches from tag edges/values).

## **Action Service:** executes tasks emitted by stored procedures on attribute changes; SignalR push plus scheduled polling for resilience.

## Original OPS Document

## **Web Tier:** IIS-hosted APIs—**/ws** (core), **/int** (integration config), **/proxy** (partner adapters)—with IIS/Windows auth and optional **OAuth2** (Azure).

## Original OPS Document

## **Deployment Model**

## A standard, repeatable rollout uses a **four-server** topology: **IP.21 historian**, **DB & core services**, **web tier & UI**, and an **all-in-one dev/support** box for configuration, testing, and investigation. An installation kit accelerates baseline installs; optional steps add file-based integrations and Data Warehouse ETL.

**Introduction**

This document introduces **OPS**, a modular Manufacturing Execution System (MES) that connects shop-floor events to business outcomes. OPS turns historian signals and operator input into structured production data (batches, steps, movements), synchronizes with enterprise systems (e.g., ERP, LIMS), and provides real-time visibility, traceability, and performance insights such as **OEE**.

**Who this is for.** Plant IT, MES/automation engineers, system integrators, and administrators responsible for deploying, configuring, and operating OPS across one or more sites.

**What you’ll learn.**

* The core architecture of OPS: web tier (APIs/UI), Windows services (**Integration Service**, **Action Service**), and data layer (**OPSDB**, **Integration Config DB**, and the **OPS Data Warehouse**).
* How OPS integrates with historians (e.g., IP.21), SCADA/OPC UA, weighbridge, Harford, ERP, and LIMS via secure Web APIs.
* Client options: the legacy **WPF** desktop client and the **HTML5** browser client (Angular/PrimeNG).
* How OEE and other KPIs are captured, stored, and reported.
* Deployment at a new site, including the recommended four-server topology and environment setup.

**Out of scope.** Detailed SOPs for plant operations, vendor-specific historian administration, and customer-specific customizations are referenced but not exhaustively covered here.

## Summary

This document describes the hierarchical framework of the 5C Pyramid in manufacturing, highlighting how integrated information systems, from sensors and actuators at the field level to MES and LIMS at the management level, enable efficient, visible, and high-quality production processes. Each level builds on the previous, ensuring seamless data flow and control from the shop floor up to business operations.

**Concept: Manufacturing Execution Solution (MES) within the 5C Pyramid**

Manufacturing industries rely on integrated information systems to ensure efficiency, visibility, and quality across production processes. One commonly used framework to describe these systems is the **5C Pyramid**, which illustrates the hierarchy of automation and information flow in modern manufacturing.

**Level 1 – Field Level**

At the base of the pyramid lies the **field level**, where data is generated. This includes **sensors, actuators, cables, and the physical infrastructure** of the plant. These elements capture real-time process data such as temperature, pressure, speed, and equipment conditions.

**Level 2 – Control Level**

The next layer is the **control level**, typically managed by **PLCs (Programmable Logic Controllers)** and low-level protocols. This layer ensures precise machine control and execution of process logic, acting as a bridge between raw data from sensors and higher-level decision-making.

**Level 3 – Supervisory Level**

The **supervisory level** includes systems such as **SCADA (Supervisory Control and Data Acquisition)** and **HMI (Human-Machine Interface)**. This layer enables operators and engineers to monitor, visualize, and intervene in real-time production processes.

**Level 4 – Management Level**

At this stage, the **Manufacturing Execution System (MES)** plays a central role. MES solutions connect the shop floor with business operations, coordinating scheduling, production orders, quality management, and compliance. In some cases, **LIMS (Laboratory Information Management Systems)** also support this level to ensure product quality and regulatory requirements.

**Level 5 – Enterprise Level**

At the top sits the **enterprise level**, powered by **ERP systems (e.g., SAP, Oracle)**. ERP integrates enterprise-wide functions such as finance, supply chain, procurement, and logistics, aligning production with overall business strategy.

**Role of MES in the 5C Pyramid**

MES acts as the **central nervous system** of manufacturing operations, bridging the gap between the real-time data of Levels 1–3 and the enterprise planning of Level 5. By providing visibility, traceability, and optimization across production, MES ensures that decision-making at all levels is data-driven and aligned with business goals.

**OPS Architecture**

OPS has a modular architecture aligned with the 5C pyramid levels.

**Control Level**

At the control level, OPS connects with plant-floor devices and PLCs to capture real-time production data.

**Historian**

OPS includes a **Historian** component (specifically, IP.21), which acts as both the process historian and SCADA. By storing time-series production data and providing graphical visualization, OPS offers customers a **monitoring system** for real-time process insight.

**Database Server**

OPS uses a **dedicated database server** that stores all critical data, including:

* Integration service data (connections with external systems)
* Action service data (execution of MES functions and workflows)

**Web Server**

OPS also has a **dedicated web server** that manages:

**Web APIs for System-to-System Communication**

OPS uses **Web APIs (Application Programming Interfaces)** as the standard way to exchange data between different software systems.

In manufacturing environments, OPS relies on Web APIs to connect with systems such as:

**1. ERP Systems (e.g., SAP)**

* **What ERP does**: ERP (Enterprise Resource Planning) systems like SAP handle the business side of operations like planning production orders, managing inventory, procurement, and logistics.
* **How OPS communicates with ERP**:
  + OPS receives **production orders** or work instructions from SAP.
  + OPS sends back **real-time production data** (e.g., how many units were made, quality status.
  + This ensures that the business system (ERP) always has an accurate view of what’s happening on the shop floor.

**2. LIMS (Laboratory Information Management System)**

* **What LIMS does**: LIMS manages laboratory test results, quality checks, and compliance data. In industries like food (cheese manufacturing), lab results are critical for ensuring safety and product quality.
* **How OPS communicates with LIMS**:
  + OPS can **send production batch information** to LIMS so that lab tests are linked with the correct product.
  + Once the lab completes tests (e.g., confirming that cheese meets quality standards), LIMS sends results back through the API to OPS.
  + OPS then uses this data to **approve or block a batch** before it moves further in the process.

### 3.Harford

particularly focused on **error prevention, quality management, and production efficiency**.

* Prevent errors on the shop floor (e.g., wrong labels, wrong raw materials).
* Ensure compliance with industry standards (food, beverage, pharma, etc.).
* Improve Overall Equipment Effectiveness (OEE).
* Provide real-time visibility into production performance.

**Key Features**:

* **Line Performance Monitoring** (track efficiency, downtime, bottlenecks).
* **Quality Assurance** (automated checks to prevent non-conformance).
* **Recipe & Batch Control** (ensuring the right process every time).
* **Traceability** (full audit trail of materials, processes, and operators).

4.Weighbridge

It records the **gross weight**, **tare weight** (empty truck), and calculates the **net weight** of goods (raw materials coming in or finished products going out

**Raw Material Intake**

* MES captures weighbridge data when raw materials (e.g., milk for cheese production) arrive.
* The system validates delivery weight against purchase orders from ERP.
* This ensures that the correct quantity is received and logged.

**Finished Goods Dispatch**

* When trucks leave with finished goods, the weighbridge provides exact shipment weights.
* MES records this against sales or delivery orders (via ERP integration).
* This ensures traceability and accurate billing.

**Traceability & Compliance**

* Weighbridge data linked into MES ensures **end-to-end traceability** (what came in, what went out).
* Provides **audit-ready records** for compliance in industries like food, pharma, and chemicals.

 **Integration with Other Systems**

* MES typically integrates weighbridge data with **ERP (e.g., SAP)** for financial reconciliation.
* It can also link with **LIMS** if weight impacts batch quality calculations (e.g., raw ingredient proportions).

**3. Other Peer MES-Level Systems**

* OPS also exchanges data with other MES or integration platforms at the same level (Level 4). For example:
  + Sharing machine performance data with a specialized analytics system.
  + Receiving scheduling input from a planning application.
  + Sending KPIs (Key Performance Indicators) to a reporting dashboard.

**Why Web APIs are Important**

* **Standardized communication**: Different systems (ERP, LIMS, SCADA, Weighbridge and Harford) often use different technologies. APIs provide a common “language” so they can share data reliably.
* **Real-time updates**: OPS can immediately notify ERP or LIMS when something changes (e.g., batch completed, test results ready).
* **Bi-directional flow**: Data doesn’t just go up (from shop floor to ERP); it also comes down (e.g., new orders from ERP, test approvals from LIMS).
* **Scalability**: New systems can be added more easily without redesigning OPS — just connect through another API.
* **Web APIs** for system-to-system communication (e.g., SAP, ERP, or other MES-level systems)
* **SignalR** for pushing live updates to user interfaces when changes occur in the database
* Reporting services for sharing operational data

**OPS Web UI (client application)**

OPS provides a **modern Web UI built on HTML5**, which means it can run smoothly in any standard web browser and is fully compatible with laptops, tablets, and mobile devices.

In addition, users can interact with OPS through a **client application**. The client communicates with the web server via the Web API, allowing users to **view, input, and exchange data** seamlessly with other connected systems.

**Historian (IP.21)**

**Purpose**: Centralized capture and retention of time-series data from multiple sources (sensors, PLC signals, and other systems).

* **Primary technology**: **AspenTech IP.21** service for ingesting and recording data; stored as tag-based time series in a NoSQL/time-series store.
* **Data model**:
  + A **tag** represents a value
  + Each record = **timestamp + value**
* **Connectivity & extensibility**:
  + Native use of **IP.21**, with built-in integration to **Wonderware** and **OPC UA**.
  + Connector framework allows rapid addition of new historians, provided they expose **tag/time-series (timestamp/value)** structures.
* **Visualization**: Beyond storage, OPS leverages the **IP.21 graphics/synoptics** to display real-time data for monitoring.
* **AspenTech** IP.21 historian is a common data backbone we can integrate with for time-series capture and visualization, while remaining vendor-agnostic for other historians.

**Databases Overview**

**OPS Database (SQL Server)** — the live, transactional store for operational data (batches, movements, users/roles, system states, reasons, audits, and reporting procedures).

**Integration Config Database** — holds all integration and system configuration for new plants. It lets teams configure and validate OPS integrations in dev/test and then promote to production environment without interrupting or modifying operational data in the OPS Database. This separation enables safer changes, parallel work, and independent.

**1) OPS Database (SQL Server – operational data)**

* **Purpose:** The OPS Database (SQL Server) is the live, transactional backbone of the OPS MES, recording and governing day-to-day operations by maintaining data integrity.
* **Contains:** Batches, movements, users, roles/privileges, system states, reason codes, audit trails, and stored procedures/views for reports and actions.
* **Usage:** Read/write by OPS runtime and Action Service; source for dashboards, reports, and traceability.

**2) Integration Config Database (configuration catalog)**

The **Integration Config Database** isolates all *configuration and integration metadata* from live MES operations, enabling safe, repeatable deployments for new plants without touching operational records.

**Purpose**

* Centralize **interface configuration** (what to connect, how, and when) separate from transactional data.
* Support **Dev → Test/UAT → Prod** promotion via backup/restore, enabling validation before go-live.
* Allow **parallel work**: teams can refine mappings and connectors while production runs uninterrupted.

**OPS Data Warehouse (Star Schema)**

**Purpose**  
The OPS Data Warehouse is a separate analytics database optimized for **reporting and BI performance**. It flattens complex OPS operational structures into **fact tables** and **dimension tables**, enabling fast, consistent queries without impacting the transactional OPS database.

**Design**

* **Dimensions (keys & context):** Hold business keys and attributes—e.g., **Unit**, **Batch**, **Product**, **Time**, **Equipment/Line**, **Location**, **Reason**, **User/Role**.
  + Preserve original source identifiers (e.g., **unit\_key**, **origin\_key**, **batch\_key**) alongside surrogate keys for lineage and joins.
* **Facts (events & measures):** Denormalized, query-friendly tables that store metrics and amounts—e.g., **Production**, **Movements/Consumption**, **Quality Outcomes**, **Tasks**, **OEE**.

**Why a warehouse?**

* The OPS operational model is rich and flexible, but some analytics require multiple joins and complex filters.
* **Fact tables** reorganize that data into **flat, analytics-ready structures**, so BI tools run **faster and more predictably**.
* Supports **greater granularity** where needed. For example, **OEE** is stored as **aggregates in OPSDB**, while the warehouse retains **max-granularity event data** for deep-dive analysis.

**Benefits**

* **Performance:** Sub-second dashboards and ad-hoc queries.
* **Separation of concerns:** Heavy analytics won’t slow live MES operations.
* **Consistency:** Stable schemas and conformed dimensions across reports.
* **Traceability:** Source keys + audit-friendly ETL for data lineage.

**ETL/Load (high level)**

* Incremental loads from OPSDB to facts/dimensions, with **surrogate keys**, late-arriving dimension handling, and (where appropriate) **SCD Type 2** to track historical attribute changes.

**Overall Equipment Effectiveness**

OEE is a single percentage that indicates how effectively a line or machine is producing versus its full potential. It blends three factors—Availability, Performance, Quality—into one score. In plain terms: the higher the OEE, the more time your assets are making good product at the right speed.

Quick example (dairy filter): If a milk/cheese filter runs 80% of the planned time, it’s earning money 80% of the time; the missing 20% is lost to stops or slow running. OEE helps you see *why* that 20% is lost and where to fix it.

 **Models/UI:** OEE timeline for **uptime/downtime**; Operator Log (Excel-like) and Batch Grids for context.

 **Data sources:** Historian/SCADA and shop-floor signals via **Integration Service**.

 **Planned vs. unplanned:** Calendars mark planned stops (e.g., CIP, changeover) vs unplanned (failures), which drives the OEE math.

 **Editing & reasons:** Operators can select events, change reasons, and record rejects/slow running.

 **Dashboards:** BI shows **OEE%**, total up/down time, top loss reasons, trends by shift/line/SKU.

 **Storage:** **OPSDB** keeps required aggregates for daily use; **OPS Data Warehouse** stores **max-granularity events** for deep analysis.

 **Event timeline:** OPS captures **uptime/downtime** as timestamped events from Historian/SCADA and shop-floor signals.

 **Planned vs unplanned:** Calendars/schedules distinguish planned stops (e.g., CIP, changeover, night shutdown) from unplanned (e.g., breakdowns). This classification feeds the OEE math.

 **Reason codes & editing:** Operators (with permissions) can select events, **edit reasons**, and add details (rejects, slow-run, blockage, label fault, etc.) to improve data quality.

 **Dashboards & KPIs:** BI views report **OEE%**, **total uptime/downtime**, top loss reasons, and trends by shift, line, SKU, and time.

 **Data storage pattern:**

* **OPSDB** holds the **aggregated** OEE needed for daily operations and decisioning.
* The **OPS Data Warehouse** stores **max-granularity events** for deep analysis and fast BI.

**Integration and Action Services**

The OPS system employs two core Windows services: the Integration Service and the Action Service.

**Integration Service:** transfers data from the Historian database to the OPS database based on configurations defined in the InterConfig Catalog. To support different data types (e.g., production batches, movements, OEE, video files), multiple instances of this service can be deployed concurrently. Each instance is uniquely identified by a Service ID parameter, which dictates the specific configuration subset it executes from the shared catalog. For example, one service instance (Service ID=1) can be configured for production batches, while another (Service ID=2) handles OEE, allowing them to operate in parallel without conflict.

**1. Integration Service**

* **Type:** Windows Service
* **Function:** Integrates data from the Historian database into the OPS database.
* **Configuration:** Governed by the InterConfig Catalog.
* **Multi-Instance Capability:** Multiple service instances can run in parallel, each dedicated to a specific data type (e.g., production batches, OEE, video files).
* **Service ID:** A unique identifier parameter that binds a service instance to its specific configuration within the shared InterConfig Catalog.

**Action Service** executes tasks queued in the OPS database. These tasks are generated by stored procedures, which are automatically triggered by changes to attribute instances in OPS. The results of these procedures are logged as tasks in a dedicated OPS table. The Action Service monitors this table, retrieves new tasks upon notification, and executes them.

Furthermore, the system incorporates an **OPS Data Warehouse**, a relational database structured with dimension and fact tables for advanced reporting and analytics.

* **Type:** Windows Service
* **Function:** Executes tasks from a queue in the OPS database.
* **Task Origin:** Tasks are generated by stored procedures triggered by changes to OPS attribute instances.
* **Workflow:**
  1. A stored procedure executes and returns one or more tasks.
  2. Tasks are written to a dedicated table in the OPS DB.
  3. A notification is sent to the Action Service.
  4. The service reads and executes the available tasks.
* **The Integration Service** moves data from our Historian system into the main OPS database. We can run several of these services at the same time for different purposes (like one for production data and another for equipment efficiency data). They are kept separate by a unique Service ID that tells each one what specific job to do.
* **The Action Service** performs automated tasks. These tasks are created automatically by the OPS system when specific data changes. The Action Service constantly checks for new tasks and runs them one by one.

The Integration Service translates raw historian tag data into structured batch events in the OPS database. This process is defined by configurations that map tag behaviors to operational events.

Consider two source tags from an IP21 historian:

* **Tag 1 (Binary Signal):** A discrete tag with values 0 or 1.
  + **Configuration:** A value change from 0 to 1 is configured as a **batch start** event. A change from 1 to 0 is configured as a **batch end** event.
* **Tag 2 (Step Indicator):** A discrete tag with integer values (e.g., 0-4).
  + **Configuration:** Each value is aliased to a step name (e.g., 1 = "Washing", 3 = "Acid Wash", 4 = "Rinse"). Each value change triggers the creation of a new **sub-event** or step, capturing its start time, end time, and name.

**Resulting Integration Output:**  
From these two tags, the Integration Service automatically generates a complete CIP batch record in OPS, including:

* Overall batch start time (e.g., 3:09 PM)
* Overall batch end time (e.g., 3:16 PM)
* Calculated total duration
* A sequence of detailed steps with their own start/end times and names.

**Technical Specifications:**

* **Type:** Windows Service built on .NET Framework.
* **Execution:** Runs on a configurable, periodic schedule defined in an application configuration file.

**Concise Summary (Best for a High-Level Overview Doc)**

The Integration Service is configured to convert historian tag data into operational events in OPS. For example, a CIP batch can be generated from two tags:

1. A **binary tag** provides the overall batch start and end times based on state changes (0->1 for start, 1->0 for end).
2. A **discrete integer tag** defines process steps. Each value change creates a new sub-event (e.g., value 1 creates a step named "Washing").

The service calculates durations and builds a structured batch record from these signals. This .NET Framework-based Windows service runs on a configurable schedule. Its 2020 version supports multiple data sources, including IP21 (ODBC/WebAPI), Wonderware, and OPC UA.

**Action Service**

A .NET Framework-based Windows service that executes tasks asynchronously. It can be triggered instantly via SignalR notification or through its periodic scheduler.

**Process:** Actions (stored procedures) are triggered by system events, generating tasks stored in a database. The Action Service processes these tasks either from a SignalR alert or by polling the table if the connection fails.

**Current Capabilities:**

* **Create Records:** Instantiates new attribute instances within a defined batch hierarchy.
* **API Calls:** Makes HTTP requests (GET, POST, PUT) to external web services with a custom payload.

**Overview**  
The **ActionService** is a Windows service (built on .NET Framework) that executes **tasks** generated by business logic in OPS. It runs on a **scheduled polling cycle** and can also be **woken up on-demand** via **SignalR** to process new tasks immediately.

**How actions become tasks**

1. A client (or integration) **creates/updates an attribute instance** through the Web API.
2. If that attribute is configured with an **action**, OPS executes the **stored procedure** that implements the action’s logic.
3. The stored procedure **emits one or more tasks**, which are written to the **Tasks** table in OPSDB.
4. The web layer then **notifies ActionService via SignalR** that new work is available.

**Triggering & scheduling**

* **Push (real-time):** SignalR notification prompts ActionService to pick up and execute tasks immediately.
* **Pull (fallback):** If SignalR is unavailable (auth failure, dropped connection, idle timeout, etc.), ActionService continues on its **periodic schedule**, scanning the Tasks table for **unprocessed** items and executing them. This ensures **no task is lost** even without real-time push.

**Supported task types (current)**

1. **Record Attribute Instance**
   * Using identifiers (e.g., route ID, attribute ID, instance number hierarchy, value), ActionService **creates a new attribute instance** within the target batch.
2. **Call External Web API**
   * Executes an HTTP **GET/POST/PUT…** to a configured **endpoint** with a specified **payload**, enabling downstream notifications or confirmations.

**Web Service Server: Architecture and Components**

A dedicated production server hosts the core OPS web service, a .NET Framework application delivered via Internet Information Services (IIS). The service provides a RESTful CRUD API, utilizing Entity Framework as its Object-Relational Mapper (ORM) to abstract database tables into software objects.

**Authentication:**  
The API supports multiple authentication methods to accommodate different deployment scenarios:

* Standard IIS Authentication
* OAuth2 (Primarily used for Azure cloud deployments)

**API Structure:**  
The web service is architecturally divided into three distinct components, each serving a specific purpose under a common host:

1. **Core API (**/ws**):** The primary service endpoint, containing the majority of the system's business logic and data operations for the OPS application.
2. **Integration Configuration API (**/int**):** A dedicated service module that provides endpoints for managing the Integration Service's configuration. This API is exclusively used by the Configuration Tool to create, read, update, and delete integration methods and instances.
3. **Proxy API (**/proxy**):** A module designed to host custom interfaces and facilitate integrations with third-party systems (e.g., DSI). This component allows for the development of specialized endpoints without impacting the core application logic, making functionalities generally available for other clients.

**Evolution:** The web service is under active development. Recent enhancements include new modules and the generalized Proxy API to standardize and expand third-party integration capabilities.

**OPS Web Service Server — High-Level Overview**

**Purpose & Scope**

The **OPS Web Service** exposes secure, standards-based APIs for user interfaces, integrations, and automation across the OPS platform. It is designed for production use with dedicated infrastructure, clear component boundaries, and extensibility for customer/partner interfaces (e.g., **DSI**).

**Technology & Hosting**

* **Platform:** .NET Framework
* **API style:** CRUD REST APIs backed by an **ORM (Entity Framework)** — each DB table maps to a domain object.
* **Hosting:** **IIS** (Internet Information Services) on a dedicated **Web Service server** in production.
* **Delivery model:** Continuous evolution to support new OPS modules and third-party integrations.

**Authentication & Security**

* **AuthN:** IIS-backed authentication; optional **OAuth2** (used in our **Azure** deployments).
* **Transport:** HTTPS only.
* **Access control:** Role/claim-based authorization at the API level.
* **Hardening:** Standard IIS security baselines, request throttling, and logging.

**Logical Components (API Areas)**

1. **Core (“WS”)**
   * Path: /ws/...
   * **Core functionality & CRUD** over OPS operational entities (batches, movements, users/roles, reasons, states, etc.).
   * Used by OPS Web UI, client apps, and internal services.
2. **Integration Config (“/int”)**
   * Path: /int/...
   * **Configuration API** for the **Integration Config Database** (connectors, endpoints, mappings, schedules, parameters).
   * Consumed by the configuration tool to **create/edit** integration methods and instances.
3. **Proxy (“/proxy”)**
   * Path: /proxy/...
   * **Custom/partner interfaces** and adapters (e.g., **DSI**), exposing standardized endpoints for third-party systems while insulating core services.

**Typical Request Flow**

Client/UI or System → HTTPS → IIS → OPS Web Service

├─ /ws → Core OPS CRUD (EF-mapped entities)

├─ /int → Integration configuration catalog operations

└─ /proxy→ Customer/partner adapters (e.g., DSI)

**Deployment & Environments**

* **Production:** Dedicated web server for isolation and performance.
* **Cloud:** Azure deployments use **OAuth2** for SSO/modern identity.
* **Versioning:** Semantic API versioning on routes or headers to support backward compatibility.

**Extensibility & Evolution**

* New modules and integrations are added as **discrete API surfaces** under /ws, /int, or /proxy.
* The **Proxy** layer enables rapid delivery of customer-specific interfaces without impacting the Core.
* ORM mappings and DTOs follow a contract-first approach to keep APIs stable.

**Operations & Observability (high level)**

* **Monitoring:** IIS logs, application telemetry, and API metrics (latency, error rates).
* **Resilience:** Graceful error handling, retries where appropriate, and throttling to protect downstream systems.

**Client Application**

We support two client interfaces:

* **Legacy WPF Client:** A Windows-only desktop application requiring local installation.
* **Next-Gen HTML5 Client:** A web-based application built with Angular/PrimeNG, accessible from any device with a browser.

We are actively transitioning to the HTML5 client as our strategic platform. This move is motivated by the need for cross-platform support (including mobile devices), a modern user experience, and a streamlined update process that eliminates manual client installations. The web client is in ongoing development to incorporate all functionalities from the legacy system.

* **Strategic Client: HTML5 Web Application**
  + Type: Single-Page Application (SPA)
  + Technology: Angular & PrimeNG
  + Deployment: Hosted on the server; accessed via browser (No installation needed)
  + Platform: Cross-platform (Windows, macOS, Linux, iOS, Android)
  + Advantages:
    - Reaches users on any device or OS.
    - Updates are instantaneous and transparent to the user.
    - Modern development and user experience.
  + Status: In active development to achieve full feature parity.

The OPS platform provides two client applications that currently coexist: a Windows-only **WPF desktop client** and a browser-based **HTML5 Web Client**. The WPF client, built on .NET/WPF, offers the widest module coverage today and is installed from the production site at <BaseURL>/ui/install, receiving per-device updates. In parallel, the new Web Client—built with **Angular** and **PrimeNG**—runs in any modern browser on Windows, macOS, Linux, iOS, and Android, using the same OPS Web APIs (and SignalR where applicable) for live updates; while not yet at full feature parity with WPF, remaining modules are in active development. This strategic move to HTML5 enables multi-platform access, simplifies upgrades (no local install—users always get the latest version on page load), and aligns with industry preference for lightweight, web-delivered shop-floor and enterprise applications.

**ConfigTool** is a web-based administrative application built on the same HTML5 stack as the OPS Web Client. It provides a secure, role-driven UI for administrators to manage configuration without code: adjust user privileges and roles, organize application navigation (menu groups and tiles), create new attributes and units, define reason codes, add states to attributes, and configure the Integration Service. By centralizing these controls in a browser, ConfigTool streamlines setup and change management across environments while keeping operational data and day-to-day production unaffected.

**ConfigTool**

A dedicated web-based administration portal (Angular/PrimeNG) for configuring the OPS system. Key responsibilities include user access control, application menu customization, core data model management (attributes, units, reasons), and integration service setup.

**Actions — High-Level Guidelines**

* **Synchronous by design:** Actions run **immediately after** an attribute is created/updated (post-commit). Their runtime **adds directly to API latency**—slow actions can trigger client **timeouts**.
* **Keep them lightweight:** **No delays/sleeps** or heavy logic inside actions. Do the minimal checks, then **emit a task** for the ActionService to handle longer work.
* **Work from the trigger:** Always start from the **triggering attribute instance ID**, perform the needed validation/counter, and **generate tasks**—don’t perform large cascades inline.
* **Avoid loops/recursion:** Never have an action that **writes to the same attribute** (or attributes that re-trigger it) without safeguards. The “same-value” guard helps, but tiny changes (spaces, counters, commas) will **re-trigger** and can create runaway loops.
* **Add guards & idempotency:** Check prior state, compare values, use **idempotency keys/flags**, and exit if the intended update has already been applied.
* **Test safely:** Validate actions in **non-prod**, and use **peer review** before enabling in production.
* **Monitor & rollback:** Track execution times and error rates; be ready to **disable** a problematic action and revert its config quickly.

**Rule of thumb:** Do **fast validation in the action**, push **all heavy work to tasks**—and design so the same input can be processed **once** and only once.

To ensure system stability and performance, adhere to the following key guidelines when configuring actions:

**1. Prioritize Performance:**

* Actions are executed synchronously by the web service immediately after a database commit. The total execution time of an action directly impacts the response time of the triggering API call.
* **Rule of Thumb:** Actions must be designed to execute as quickly as possible. Avoid introducing deliberate delays (e.g., SLEEP commands), as this will cause API timeouts and degrade overall system performance.

**2. Prevent Recursive Loops:**

* A critical risk occurs when an action is triggered by an update to a specific attribute (e.g., ID 1000) and its resulting task then updates that same attribute, creating an infinite loop.
* **Rule of Thumb:** Implement a clear termination condition in your action logic to prevent it from re-triggering itself. Carefully audit the action's outcome to ensure it does not modify the attribute that initiated it.

**3. Implement a Peer Review Process:**

* Due to the potential for significant system-wide impact, all new or modified actions should be reviewed by a second developer or administrator before deployment.
* **Rule of Thumb:** Mandate a code review for all actions to validate their logic, efficiency, and most importantly, to identify potential recursive loops.

**Conclusion:** While powerful, actions can introduce performance bottlenecks and recursive failures if misconfigured. Their design must prioritize speed and include safeguards against loops, enforced by a strict peer review policy.

**Overview: OPS System Deployment & Architecture**

**1. System Deployment**

A new OPS site deployment is structured across a multi-server environment to ensure performance, scalability, and separation of concerns.

* **Standardized Installation:** An installation kit is available to deploy the entire OPS system in a standard configuration rapidly. This standard setup serves as the foundation for approximately 90-95% of all sites.
* **Customizable Components:** Post-installation, optional services like file-based integrations or the OPS Data Warehouse can be manually deployed to meet specific site requirements.
* **Ideal Server Architecture:** The recommended production environment utilizes four dedicated servers:
  + **Server 01 (Historian):** Hosts the AspenTech IP21 historian suite.
  + **Server 02 (Database & Core Services):** Contains the live OPS SQL Server database, Integration Service, Action Service, and reporting tools (SSRS, SSMS). This is the core production data server.
  + **Server 03 (Web Services & UI):** Hosts the web application infrastructure, including:
    - The Core Web Service API (/ws)
    - The HTML5 Client UI
    - The Integration Config API (/int)
    - The Proxy API (/proxy)
    - The ConfigTool
  + **Server 04 (Development/Support Server):** A consolidated server containing all software components (IP21, Database, Services, Web API, UI). This server is used for configuration, testing, troubleshooting, and investigation without impacting the production environment.

**2. Core Investigation Concepts**

To effectively verify configuration or investigate issues, particularly with batches, understanding the following is essential:

* **Database Structure:** The OPS database is the system of record. Proficiency in navigating its tables is required for deep analysis.
* **Batch Anatomy:** A thorough understanding of how a batch is structured within the database is critical. This includes knowing the key tables and relationships that store batch information, its events, and associated attribute data.
* **Key Information for Troubleshooting:** Successful investigation hinges on knowing what data points to examine to determine if a batch was processed correctly, if an integration failed, or if a configuration is functioning as intended.

**Deploying OPS at a New Site**

**Scope.** This section outlines how we deploy OPS on a new site, the standard server topology, and the few cases that require manual steps. Deeper topics (database internals, batch structure, troubleshooting) follow in later chapters.

**Standard installation approach**

* **Installation kit (coming soon):** Installs a **standard OPS configuration** end-to-end in minutes. This is the starting point for ~**90–95%** of sites.
* **When manual steps are needed:**
  + **File integrations** (custom, file-based)
  + **Data Warehouse** enablement (manual deploy of required stored procedures/ETL to link OPS → DW)

**Recommended 4-server topology (production)**

1. **Server-01 — Historian**
   * **Role:** AspenTech **IP.21** (full suite) hosting the plant historian.
2. **Server-02 — Database & Services**
   * **Role:** **SQL Server** with OPS catalogs (**OPSDB** + **Integration Config DB**), **ActionService**, **IntegrationService**, **SSRS**, **SSMS**.
   * **Typical paths:** D:\Services\IntegrationService1, D:\Services\ActionService, etc.
3. **Server-03 — Web Tier**
   * **Role:** **IIS** hosting **OPS Web API** (/ws), **Integration Config API** (/int), **Proxy** (/proxy), **HTML5 Web UI**, and **ConfigTool**.
   * **Typical path:** D:\WebServices\OPS.
4. **Server-04 — Dev/All-in-One**
   * **Role:** Consolidated environment for configuration, testing, and issue investigation: **IP.21**, **SQL Server**, **Action/Integration Services**, **SSRS/SSMS**, **Web API**, **HTML5 UI**, **IIS**.

**High-level deployment steps**

1. **Provision infrastructure:** Create Server-01..04 VMs, apply OS baselines, open required ports, create service accounts.
2. **Install historian:** Deploy **IP.21** on Server-01 and validate tag access.
3. **Install database & services:** On Server-02, install **SQL Server**, restore/create **OPSDB** and **Integration Config DB**, deploy **ActionService** and **IntegrationService** (set **ServiceID**s), install **SSRS** and **SSMS**.
4. **Install web tier:** On Server-03, configure **IIS**, deploy **OPS Web API** (/ws, /int, /proxy), **HTML5 UI**, **ConfigTool**; enable HTTPS and authentication (IIS/Windows or **OAuth2** for Azure).
5. **Stand up dev box:** On Server-04, deploy all components for end-to-end dry runs and troubleshooting.
6. **Configure integrations:** Point connectors (IP.21/Wonderware/OPC UA, SAP/ERP, LIMS, Harford, Weighbridge), load mappings/schedules into **Integration Config DB**.
7. **Smoke tests:**
   * Historian connectivity (tags read)
   * DB health (reads/writes, SPs, jobs)
   * Web API endpoints, auth, and **SignalR** push
   * Services running with correct **ServiceID**s
   * SSRS reports render
8. **Handover:** Document endpoints, credentials (vaulted), backup plans, and monitoring dashboards.

**Mobile Manifest**

Automate and standardize **raw-milk intake** when a tanker truck arrives, capturing only the **essential load data** from a cloud API and storing it in OPS for traceability, quality gating, and downstream processing.

**Actors & Systems**

* **Driver + Mobile Scanner**: scans a **Load Identifier** (unique ID) at gate or intake bay.
* **Cloud Intake API**: returns full load payload (farm, driver, supplier, quantities, timestamps, sampling info, etc.), keyed by **load\_id**.
* **OPS Web Service (API)**: requests just the fields OPS needs and writes them to **OPSDB**.
* **Integration Config DB**: holds mapping/validation rules (incl. **MM\*** manifest controls).

**End-to-End Flow**

1. **Arrival & Scan**
   * Driver scans the **Load Identifier** on a mobile device.
2. **Lookup**
   * OPS calls the **Cloud Intake API** with load\_id.
3. **Selective Extract**
   * From the large payload, OPS pulls **only the required fields** (minimal data principle):
     + **Keys**: load\_id (primary), optional external reference keys.
     + **Transport legs**: inbound leg details needed by OPS (start/end, source/destination, volumes).
     + **Core context**: farm/supplier IDs, driver ID (as references), timestamps, basic quantities.
     + **Sampling flags/IDs** (if needed for LIMS linkage).
4. **Persist to OPS**
   * OPS writes the intake record to **OPSDB**, establishing material genealogy and readiness for weighbridge/OEE/quality workflows.
5. **Downstream Use**
   * Data becomes available to batch creation, quality sampling, and ERP posting via existing OPS integrations.

**Data Model Notes**

* **Primary key**: load\_id (unique per tenant/site).
* **Transport legs**: stored as child records for the intake header; drive movements and scheduling.
* **Reference linking**: farm/supplier/driver kept as external references; enrich later if needed.

**Configuration & Control**

* **Manifest-controlled fields (MM\*)**: Any field prefixed **MM** is governed by a **manifest** in the Integration Config DB (enable/disable, required/optional, validation rules, defaulting).
* **Mapping & validation**: All transformations and field selections live in config, not code, enabling promotion **Dev → Test → Prod** via backup/restore.

**Security & Tenancy**

* **Auth**: OPS authenticates to the Cloud Intake API using per-client credentials (OAuth2/API key).
* **Tenant resolution**: Requests execute in the caller’s tenant/site context; load\_id is treated as **unique within tenant**.

**Reliability & Idempotency**

* **Idempotent writes**: Upserts by load\_id prevent duplicates on retries.
* **Fallback**: If the push/SignalR path is unavailable, scheduled polling can re-query pending loads.
* **Audit trail**: Store request/response metadata (hash, timestamp, status) for traceability.

**Error Handling (high level)**

* **Validation failures** (missing required manifest fields) → reject with clear reason; keep a reject log.
* **Upstream unavailability** → retry with back-off; alert if SLA breached.
* **Partial payloads** → accept minimal viable record (per manifest) and flag for enrichment.

**What We Don’t Store**

To reduce risk and clutter, OPS **does not persist** nonessential fields returned by the API (e.g., full driver bio or verbose farm metadata) unless explicitly enabled in the manifest.