# Opening CIP: An OpenAPI for Describing CIP-Enabled Products

Greg Majcher
Principal Application Engineer
Rockwell Automation

Joakim Wiberg
Director of Technology
ODVA

Presented at the ODVA
2026 Industry Conference & 24th Annual Meeting
October 21, 2026
Vilamoura, Portugal

## Abstract

The Common Industrial Protocol (CIP) is widely used in industrial automation to enable interoperable communication between controller and field devices from multiple vendors. As the number and diversity of CIP-enabled products continue to grow, the need for standardized, machine-readable mechanisms to describe device capabilities, characteristics, and related metadata has become increasingly important. Such mechanisms can simplify integration with engineering tools, digital repositories, and emerging data-driven applications.

This paper presents the development of an API based on the OpenAPI Specification to describe products that implement CIP. The proposed approach provides a consistent and structured method for accessing product information through a standardized REST interface, enabling automated interaction with product data across tools and platforms.

The paper introduces a proposed OpenAPI definition designed to represent key product attributes, including device identity, supported features, and associated documentation. The definition is intended to promote interoperability while simplifying access to device information for system integrators, engineers, application developers, and data scientists.

Several example use cases are presented to demonstrate how the proposed OpenAPI definition can be applied in practical scenarios. These examples illustrate how standardized API access can support activities such as device discovery, feature identification, and large-scale data harvesting.

The results demonstrate how an OpenAPI-based approach can improve accessibility, automation, and interoperability in managing information related to CIP-enabled products and supporting modern industrial engineering workflows.

 ©2026 ODVA, Inc.

## Keywords

Common Industrial Protocol (CIP), EtherNet/IP, OpenAPI Specification, REST API, Device Identity, Device Discovery, Interoperability, Industrial Automation, Electronic Data Sheet (EDS), Data Harvesting, Predictive Maintenance, Digital Twin

## Definition of terms

**API (Application Programming Interface)** — A defined interface through which one software system can interact with another. An API specifies the operations that are available, the inputs they accept, and the outputs they return, allowing applications to communicate without knowledge of each other's internal implementation.

**HTTP / HTTPS** — Hypertext Transfer Protocol (HTTP) is the foundational communication protocol used for data exchange on the web. HTTPS is the secure variant, in which all communication is encrypted and authenticated using TLS (Transport Layer Security).

**REST (Representational State Transfer)** — An architectural style for building networked applications. A REST API exposes resources through URLs and uses standard HTTP methods such as GET to interact with them. REST APIs are stateless, meaning each individual request contains all the information needed to process it — the server does not retain session state between requests.

**URL (Uniform Resource Locator)** — A standardized address used to identify a resource on a network. In the context of REST APIs, each resource — such as a device or an attribute — is identified by a unique URL that clients use to send requests.

**JSON (JavaScript Object Notation)** — A lightweight, human-readable data interchange format widely used in REST APIs. JSON [1] represents data as key-value pairs and arrays, and is natively supported by virtually all modern programming languages and platforms.

**Endpoint** — A specific URL exposed by a REST API that a client can send requests to. Each endpoint represents a resource or operation, and responds to one or more HTTP methods such as GET.

**OpenAPI Specification** — A standardized, language-agnostic format for describing REST APIs. An OpenAPI [2] definition formally specifies the endpoints, parameters, request and response structures, and associated metadata of an API in a machine-readable document, enabling automatic code generation, documentation, and validation.

**curl** — A free, open-source command-line tool for transferring data using URLs. It supports a wide range of protocols including HTTP and HTTPS, and is commonly used to manually send API requests and inspect responses. curl is available on Windows, macOS, and Linux, and requires no installation on most modern operating systems.

**CMMS (Computerized Maintenance Management System)** — Software used to manage and track maintenance activities, work orders, and physical assets in industrial and facility environments.

**Z-score** — A statistical measure that expresses how far a data point deviates from the mean of a dataset, in units of standard deviation. A high z-score indicates that a value is unusual relative to recent historical data, and is commonly used as a simple basis for anomaly detection.

## Introduction

The increasing adoption of CIP-enabled devices across industrial environments has led to a growing diversity of products from a wide range of vendors. While CIP provides a robust foundation for

 ©2026 ODVA, Inc.

interoperable communication, the information describing these products — their capabilities, characteristics, and associated metadata — remains largely inaccessible in a standardized, machine-readable form. This creates friction for system integrators, engineers, and application developers who must rely on manual processes or proprietary tools to retrieve and work with product information.

CIP already provides a consistent, machine-readable model for describing device capabilities — but retrieving that information requires a CIP-capable stack and tooling that understands CIP messaging. Figure 1 illustrates a typical scenario: field devices, a PLC, and an HMI on the OT network, separated from the IT network by a gateway. On the IT network, a CMMS — which may run locally or in the cloud — needs data from one or more devices on the OT network to support maintenance and asset management workflows. The problem is that this software has no understanding of CIP and no CIP stack with which to retrieve that data. This is representative of a broader set of tools — scripting languages, spreadsheet applications, business intelligence platforms — that sit on the IT side and have no native way to communicate with CIP devices at all. As industrial systems increasingly intersect with data-driven applications, digital repositories, and software toolchains built on standard web technologies, the need for a protocol-agnostic, HTTP-accessible path to this same information becomes more pressing.

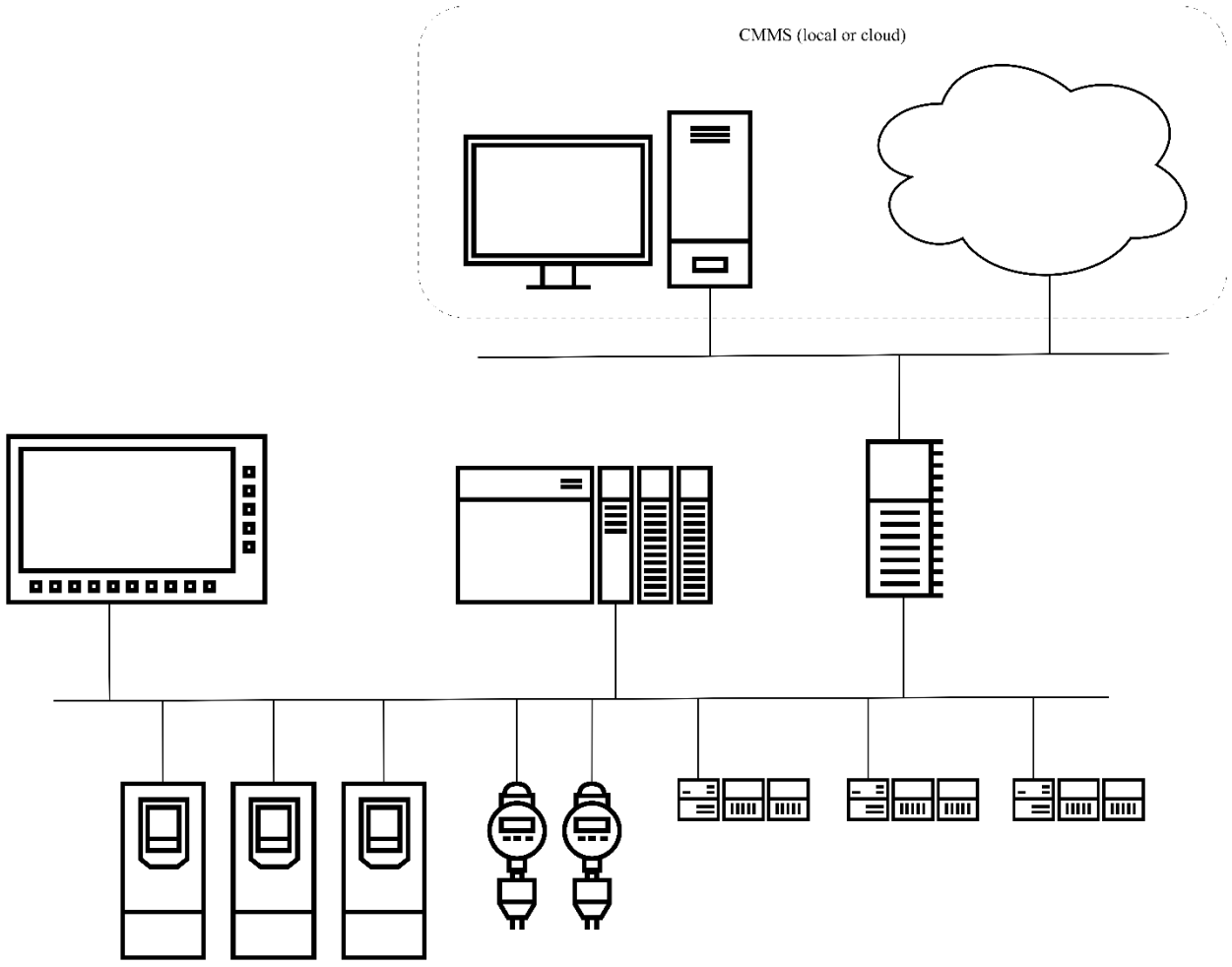



**Figure 1 A CMMS on the IT network needing device data from an OT network it has no CIP stack to communicate with**

This paper proposes an API based on the OpenAPI Specification as a standardized approach to describing CIP-enabled products. The proposed definition provides a REST interface through which key

product attributes — including device identity, supported features, and associated documentation — can be accessed in a consistent and interoperable manner. Several use cases are presented to illustrate practical applications of the approach, ranging from device discovery and feature identification to large-scale data harvesting. Section "Comparison with alternative approaches" discusses how this approach relates to existing mechanisms, including EDS files, the Extended Logical segment, i3X, OPC UA, and MQTT.

## Background and Related Work

Describing device capabilities and characteristics in a machine-readable way has been a longstanding challenge in industrial automation. Several approaches have been developed over the years to address this, each with varying degrees of standardization and applicability.

Electronic Data Sheets (EDS) have been the traditional mechanism for describing CIP device capabilities. An EDS file provides a structured, text-based description of a device's configurable parameters and communication capabilities, enabling engineering tools to configure and integrate devices. While widely adopted, EDS files are primarily designed for device configuration rather than broader product information exchange, and their utility is largely confined to specific tooling ecosystems.

Recent work presented at this conference has sought to expand on these foundations. Majcher [3] examined the challenge of making CIP device data more discoverable and understandable, with particular attention to data science use cases. That work highlighted limitations in existing online discovery mechanisms and proposed extensions to EDS files — including richer use of Object Class sections and Param entries — to expose a more complete picture of device data to clients and tools. Building on this, Majcher and Schaffner [4] addressed the broader question of metadata in CIP devices, arguing that richer and more consistently provided metadata is essential for enabling modern workflows including asset management, predictive maintenance, and data-driven optimization. That paper proposed a mechanism for devices to expose metadata online through a new Extended Logical segment, and called for EDS files to present more comprehensive device information models. Together, these contributions establish the value and necessity of structured, accessible device information — and motivate the need for a standardized, web-friendly interface through which such information can be retrieved.

Outside of industrial automation, the problem of describing APIs and services in a machine-readable way has been addressed effectively by the OpenAPI Specification (OAS). Originally derived from the Swagger framework [9], OAS has become the de facto standard for describing RESTful APIs. It provides a language-agnostic, human- and machine-readable format for defining endpoints, data models, and interaction patterns, and is supported by a broad ecosystem of tools for code generation, documentation, and testing.

The application of REST-based interfaces and OpenAPI in industrial contexts has gained significant momentum in recent years. A notable example is i3X (Industrial Information Interoperability eXchange) [5], an open, vendor-agnostic REST API specification coordinated by CESMII. A working group including Rockwell Automation, Siemens, and Inductive Automation, among others, has developed i3X to provide a common interface for accessing contextualized manufacturing data, so that application developers can code against a single, stable API surface deployable across any compliant platform. Like the API proposed in this paper, i3X is itself defined using OpenAPI, and is currently available in public beta.

However, i3X operates at the platform level, addressing interoperability across manufacturing data sources broadly. It does not address the specific need this paper targets: a standardized interface for describing individual CIP-enabled products — their identity, capabilities, and associated metadata. That gap is what the present paper addresses.

## REST APIs and the OpenAPI Specification

Modern software systems increasingly rely on web-based interfaces to exchange data in a standardized and accessible way. One of the most widely adopted patterns for building such interfaces is REST, which stands for Representational State Transfer. A REST API is an interface that allows software clients to interact with a server by sending HTTP requests — the same underlying protocol used by web browsers. Each request targets a specific resource identified by a URL, and the server responds with structured data, typically formatted as JSON. REST APIs are stateless, meaning each request contains all the information needed to process it, making them simple to implement, easy to scale, and accessible from virtually any programming language or platform. Security is also well addressed; when HTTP is replaced by HTTPS, all communication is encrypted and authenticated using TLS — the same underlying security mechanism employed by CIP Security [6] — making REST APIs suitable for use in environments where data integrity and confidentiality are required.

To introduce these concepts independently of CIP-specific detail, the following examples use a generic, widely familiar scenario before returning to CIP-enabled devices in the next section.

To illustrate, consider a hypothetical weather service hosted at api.example.com. A software client wanting to retrieve the current temperature for a given city could send an HTTP GET request to the endpoint shown in Figure 2. The URL identifies the endpoint — in this case */weather/v2/current*, representing the current weather resource — while the query *parameter ?city=Chicago* provides the input to the request, specifying the location for which the weather data should be retrieved.

Well-designed REST APIs typically include a base path that serves two purposes: scoping and versioning. The base path scopes the API to a specific domain or service, avoiding conflicts with other resources that may be available at the same address. It also includes a version identifier, ensuring that future changes to the API can be introduced without disrupting existing clients. The weather example in Figure 2 illustrates this — */weather* scopes the API to weather-related resources and */v2* identifies the API version.

```
GET https://api.example.com/weather/v2/current?city=Chicago
```

**Figure 2 Example HTTP GET request to a weather service API**

The server would respond with a structured JSON object containing the requested information, as shown in Figure 3. In the JSON response, numeric values such as temperature are represented as decimal numbers. This is a requirement of the JSON specification, which does not support hexadecimal notation. While this distinction is not relevant for a weather example, it is worth noting in the context of CIP, where hexadecimal is commonly used to express values such as Class IDs, Vendor IDs, and Device Types.

```
{
  "city": "Chicago",
  "temperature": 18.5,
  "units": "celsius",
  "condition": "windy"
}
```

**Figure 3 Example JSON response from a weather service API**

This approach is immediately accessible to any developer familiar with web technologies, requires no specialized middleware, and integrates naturally with modern software toolchains.
REST's value in this context comes from its ubiquity: because HTTP and JSON are already supported everywhere, a dedicated weather application and a general-purpose data analysis tool can consume the same interface without any translation layer.

However, a REST API on its own does not describe itself. A developer integrating with an unfamiliar API must rely on external documentation to understand what endpoints are available, what parameters they accept, and what responses they return. This is where the OpenAPI Specification adds significant value. OpenAPI is a standardized, machine-readable format for describing REST APIs. An OpenAPI definition is a document — typically written in JSON or YAML [7] — that formally specifies every endpoint, its parameters, the structure of requests and responses, and any associated metadata. It is both human-readable and processable by tools, enabling automatic generation of client code, interactive documentation, and validation of API requests and responses.

Returning to the weather example, an OpenAPI definition for the current weather endpoint would formally declare that the endpoint accepts a city name as a query parameter, and returns a response conforming to a defined schema that includes temperature, units, and condition. A simplified excerpt of such a definition is shown in Figure 4.

```yaml
paths:
  /weather/current:
    get:
      summary: Get current weather for a city
      parameters:
        - name: city
          in: query
          required: true
          schema:
            type: string
      responses:
        '200':
          description: Current weather data
          content:
            application/json:
              schema:
                type: object
                properties:
                  city:
                    type: string
                  temperature:
                    type: number
                  units:
                    type: string
                  condition:
                    type: string
        '404':
          description: City not found
```

**Figure 4 OpenAPI definition excerpt for a weather service endpoint**

The '200' key in the response definition refers to the HTTP status code returned when the request succeeds — status code 200 is the standard HTTP response for a successful GET request, commonly known as "200 OK". Alongside the 200 OK response, APIs typically also define responses for cases where a request cannot be fulfilled. A 404 Not Found response, for example, indicates that the requested resource does not exist — not that something went wrong, but simply that the server has no data at the

specified location. Defining these response codes explicitly in the OpenAPI definition ensures that clients know what to expect in all cases, not just the successful one.

A reader familiar with REST APIs may notice that Figure 4 shows the path as */weather/current* rather than */weather/v2/current* as used in Figure 2. This is because the version identifier is not part of the individual path definition in an OpenAPI document. Instead, it is declared once in the servers block, which specifies the base URL against which all paths are resolved. For the weather example, the servers block would declare the base URL as *https://api.example.com/weather/v2*, as shown in Figure 5, and OpenAPI tooling concatenates this with each path to form the full URL. The path /weather/current therefore resolves to https://api.example.com/weather/v2/current, as shown in Figure 2, without the version appearing in any individual path definition.

```
servers:
  - url: https://api.example.com/weather/v2
    description: Example weather API
```

**Figure 5 OpenAPI servers block declaration for the weather service API**

This separation of concerns is a deliberate design choice in OpenAPI. Keeping the version out of the path definitions means that all paths can be updated to a new version in a single place — by changing the servers block — without modifying any individual endpoint definition.

Any tool that can read this OpenAPI document can immediately understand how to interact with the API, what data to expect, and how that data is structured — without any manual intervention. This is particularly powerful when many different clients need to integrate with the same service, as the definition serves as a precise, unambiguous contract between the API provider and its consumers.

Together, REST and OpenAPI provide a foundation that is simultaneously accessible to web and enterprise developers, compatible with modern tooling ecosystems, and expressive enough to describe complex information structures. The interface proposed in this paper is built on this foundation.

# Proposed OpenAPI Definition

## Design goals and principles

The proposed OpenAPI definition is designed around three core principles: interoperability, simplicity, and extensibility. Interoperability is the primary motivation — the definition provides a common interface that any client, regardless of vendor or platform, can use to retrieve product information from a CIP-enabled device. Simplicity guides the scope of the interface: the definition exposes a well-bounded set of product attributes through a small number of clearly defined endpoints, avoiding unnecessary complexity while remaining practically useful. Extensibility ensures that the definition can accommodate the diversity of CIP-enabled products, allowing vendors to supplement the core set of attributes with product-specific information without breaking compatibility with generic clients.

The interface is hosted directly on the device itself, served over HTTPS alongside or in proximity to the device's existing communication stack. This approach is consistent with the trend toward embedding web-capable interfaces in industrial devices and avoids the need for an intermediary repository or proxy. Without any additional infrastructure, a client that can reach the device over a network can retrieve structured product information using standard HTTP tooling.

The definition is expressed as an OpenAPI document conforming to OpenAPI Specification version 3.1. All responses are returned as JSON. The API is read-only — it exposes product information but does not accept writes — which simplifies implementation and reduces the attack surface. Where TLS is available, all communication is encrypted, consistent with CIP Security practices.

## API Structure and Endpoints

The proposed definition organizes product information into four resource areas, each accessible through a dedicated endpoint: device identity, supported object classes and their attributes, live attribute values, and device documentation.

All field names in API responses follow snake_case convention, where words are written in lowercase and separated by underscores (e.g., vendor_id, product_name, device_type). This convention is widely adopted in REST APIs and is natively supported by most programming languages and data processing tools. Field names and URL paths are case-sensitive, consistent with standard REST practice; clients should use the exact casing shown in the API definition.

All endpoints are served under the base path */cip/v1*, where /cip scopes the API to CIP-related resources and avoids conflicts with other services that may be hosted on the same device, and /v1 identifies the version of the API. Versioning the base path ensures that future revisions of the API can be introduced without breaking existing clients, which can continue to use */cip/v1* while newer clients adopt /cip/v2 or later versions. As described in the previous section, the base path is declared once in the servers block of the OpenAPI definition rather than repeated in every path definition, as shown in Figure 6. The full URL for any endpoint is then formed by concatenating the base URL with the path, so */identity* resolves to *https://{device_ip}/cip/v1/identity*, where {device_ip} is the IP address of the device.

```
servers:
  - url: http://{device_ip}/cip/v1
    description: Device-hosted API
    variables:
      device_ip:
        default: 192.168.1.1
        description: IP address of the CIP device
```

**Figure 6 OpenAPI servers block declaration for the CIP device API**

### Device Identity

The */identity* endpoint returns the core identifying attributes of the device, sourced directly from the Identity Object (Class 0x01) [8].

The response includes the seven mandatory instance attributes defined by the Identity Object — Vendor ID (instance attribute 1), Device Type (instance attribute 2), Product Code (instance attribute 3), Revision (instance attribute 4), Status (instance attribute 5), Serial Number (instance attribute 6), and Product Name (instance attribute 7) — along with three additional instance attributes: Assigned_Name (instance attribute 15), Catalog Number (instance attribute 21), and Vendor Name (instance attribute 31). The optional instance attributes are omitted from the response entirely when not supported by the device. Clients should therefore treat their absence as an indication that the device does not implement those attributes, rather than as an error condition. The Assigned_Name attribute uses the CIP STRINGI data type, which supports internationalized strings. In the JSON response this is represented as an array of objects, each containing a language code and the corresponding string value.

Revision is represented as a nested object with major and minor integer fields, reflecting the two-byte structure of the CIP attribute.

The Status instance attribute is represented as a DeviceStatus object, with the raw status word decomposed into individually named boolean fields for each defined bit. This avoids requiring clients to perform bitwise operations and makes the device state directly readable. The six bit fields — owned (bit 0), configured (bit 2), minor_recoverable_fault (bit 8), minor_unrecoverable_fault (bit 9), major_recoverable_fault (bit 10), and major_unrecoverable_fault (bit 11) — are all required. The two

extended device status nibbles (bits 4–7 and bits 12–15) are returned as optional human-readable strings when the device provides a meaningful description; they are omitted otherwise.

A representative response from a device returning the instance attributes is shown in Figure 7. As with all numeric values in JSON, values such as vendor_id, device_type, and product_code are expressed in decimal, even though they are commonly written in hexadecimal in CIP documentation and configuration tools.


```
{
  "vendor_id": 1,
  "device_type": 12,
  "product_code": 65,
  "revision": {
    "major": 3,
    "minor": 1
  },
  "status": {
    "owned": true,
    "configured": true,
    "minor_recoverable_fault": false,
    "minor_unrecoverable_fault": false,
    "major_recoverable_fault": false,
    "major_unrecoverable_fault": false,
    "extended_device_status": "No I/O connection established"
  },
  "serial_number": 16909060,
  "product_name": "1756-L84ES ControlLogix 5580",
  "catalog_number": "1756-L84ES",
  "vendor_name": "Rockwell Automation",
  "assigned_name": [
    {"language": "eng", "value": "Drive train - sub assembly 3"},
    {"language": "spa", "value": "Tren de transmisión - subconjunto 3"},
    {"language": "deu", "value": "Antriebsstrang - Unterbaugruppe 3"}
  ]
}
```


**Figure 7 JSON response from GET /identity for a device supporting all ten instance attributes**

The corresponding OpenAPI path definition formally declares the structure of this response, as shown in Figure 8.

```
paths:
  /identity:
    get:
      tags: [Identity]
      summary: Get device identity
      operationId: getIdentity
      description: >
        Returns the identity attributes of the device, sourced from the
```

 ©2026 ODVA, Inc.

```
        Identity Object (Class 0x01).
      responses:
        '200':
          description: Device identity
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/DeviceIdentity'
```

**Figure 8 OpenAPI path definition for GET /identity**

The DeviceIdentity and DeviceStatus schemas, defined in the components section of the OpenAPI document, type and annotate each field with its corresponding instance attribute number, providing an explicit and machine-readable bridge between the REST representation and the underlying CIP data model.

The OpenAPI definition uses schemas to formally describe the structure of each response. Figure 9 shows the DeviceIdentity schema, which defines the fields returned by the /identity endpoint, their data types, and which fields are mandatory. Each field corresponds directly to a CIP Identity Object instance attribute, with the attribute number noted in the description to make the mapping explicit.

The status field is not defined inline within DeviceIdentity. Instead, it uses a $ref — a reference to a separate schema named DeviceStatus. This keeps the definition modular and avoids repetition; DeviceStatus can be defined once and reused wherever needed. The $ref mechanism is a standard feature of the OpenAPI Specification and works similarly to a pointer — when a client or tool processes the DeviceIdentity schema, it follows the $ref to DeviceStatus to resolve the full definition of the status field. The contents of DeviceStatus are omitted from Figure 9 for brevity but are described in the following section.

```
components:
  schemas:

    DeviceIdentity:
      type: object
      required:
        - vendor_id
        - device_type
        - product_code
        - revision
        - status
        - serial_number
        - product_name
      properties:
        vendor_id:
          type: integer
          description: "CIP Identity Object attr 1 – ODVA-assigned Vendor ID"
          example: 1
        device_type:
          type: integer
```

```
          description: "CIP Identity Object attr 2 – CIP Device Type code"
          example: 12
        product_code:
          type: integer
          description: "CIP Identity Object attr 3 – Vendor-assigned Product Code"
          example: 65
        revision:
          type: object
          description: "CIP Identity Object attr 4 – Firmware revision"
          required: [major, minor]
          properties:
            major:
              type: integer
              example: 3
            minor:
              type: integer
              example: 1
        status:
          $ref: '#/components/schemas/DeviceStatus'
        serial_number:
          type: integer
          format: int32
          description: "CIP Identity Object attr 6 – Device serial number"
          example: 16909060
        product_name:
          type: string
          description: "CIP Identity Object attr 7 – Human-readable product name"
          example: "1756-L84ES ControlLogix 5580"
        catalog_number:
          type: string
          description: "CIP Identity Object attr 21 – Vendor catalog number"
          example: "1756-L84ES"
        vendor_name:
          type: string
          description: "CIP Identity Object attr 31 – Vendor name string"
          example: "Rockwell Automation"
        assigned_name:
          type: array
          description: "CIP Identity Object attr 15 – Internationalized device name
(STRINGI). Represented as an array of language/value pairs. Omitted if not
supported."
          items:
            $ref: '#/components/schemas/StringIEntry'
          example:
            - language: "eng"
```

```
            value: "Drive train - sub assembly 3"
          - language: "spa"
            value: "Tren de transmisión - subconjunto 3"
          - language: "deu"
            value: "Antriebsstrang - Unterbaugruppe 3"
```

**Figure 9 OpenAPI schema definition for DeviceIdentity**

### CIP Object Discovery

A fundamental aspect of interacting with a CIP-enabled device is knowing which objects it implements. Two endpoints address this: */objects* provides a concise inventory of implemented object classes, while */objects/instances* extends that inventory with the instance numbers that exist on the device for each class.

*GET /objects* returns an array of CIPObjectSummary entries, each containing a class_id integer and a class_name string. The list reflects only the object classes implemented by the device. This makes the response immediately useful for discovery purposes, as a client can determine what the device supports without any prior knowledge of its profile or configuration.

The OpenAPI path definitions for these two endpoints are shown in Figure 10. Both endpoints respond to HTTP GET requests and return arrays of objects — CIPObjectSummary for */objects* and CIPObjectWithInstances for */objects/instances*. As with the /identity endpoint, the response structures are defined as named schemas and referenced using $ref, keeping the path definitions concise and the schema definitions reusable.

```
/objects:
  get:
    tags: [Objects]
    summary: List implemented CIP objects
    operationId: listObjects
    description: >
      Returns a list of all CIP Object classes implemented by this device.
    responses:
      '200':
        description: List of implemented CIP objects
        content:
          application/json:
            schema:
              type: array
              items:
                $ref: '#/components/schemas/CIPObjectSummary'


/objects/instances:
  get:
    tags: [Objects]
    summary: List implemented CIP objects with their instances
    operationId: listObjectsWithInstances
    description: >
```

```
        Returns a list of all CIP Object classes implemented by this device,
        each including the instance numbers that exist for that class.
      responses:
        '200':
          description: List of implemented CIP objects with instances
          content:
            application/json:
              schema:
                type: array
                items:
                  $ref: '#/components/schemas/CIPObjectWithInstances'
```

**Figure 10 OpenAPI path definitions for /objects and /objects/instances**

The schemas referenced by these endpoints are shown in Figure 11. CIPObjectSummary is a simple two-field object containing a class_id integer and a class_name string. CIPObjectWithInstances extends this with an instances field — an array of integers listing the instance numbers implemented by the device for that object class. Both schemas are intentionally minimal, returning only the information needed for discovery without exposing implementation details that may vary between devices.

```
    CIPObjectSummary:
      type: object
      required:
        - class_id
        - class_name
      properties:
        class_id:
          type: integer
          description: "CIP Object Class ID"
          example: 1
        class_name:
          type: string
          description: "CIP Object class name"
          example: "Identity Object"

    CIPObjectWithInstances:
      type: object
      required:
        - class_id
        - class_name
        - instances
      properties:
        class_id:
          type: integer
          description: "CIP Object Class ID"
          example: 4
        class_name:
          type: string
```

```
          description: "CIP Object class name"
          example: "Assembly Object"
        instances:
          type: array
          description: "List of instance numbers implemented by the device for this
object class"
          items:
            type: integer
          example: [1, 2, 3, 100, 101]
```

**Figure 11 OpenAPI schema definitions for CIPObjectSummary and CIPObjectWithInstances**

Where a client needs to go beyond class-level discovery and understand the instance structure of the device, *GET /objects/instances* returns the same list augmented with an instances array for each entry. Each element of the array is an instance number that exists on the device for that object class. A representative response is shown in Figure 12.


```
[
  {
    "class_id": 1,
    "class_name": "Identity Object",
    "instances": [1]
  },
  {
    "class_id": 4,
    "class_name": "Assembly Object",
    "instances": [1, 2, 3, 100, 101]
  },
  {
    "class_id": 6,
    "class_name": "Connection Manager Object",
    "instances": [1]
  }
]
```


**Figure 12 JSON response from GET /objects/instances**

Together, these two endpoints provide a structured and machine-readable foundation for device discovery, enabling engineering tools and data-driven applications to systematically map the object model of a device without relying on EDS files or manual inspection.

### Attribute Discovery and Value Retrieval

While the */objects* and */objects/instances* endpoints provide a high-level inventory of what a device implements, two additional endpoints allow clients to interrogate specific object instances in detail and retrieve live attribute values directly from the device.

*GET /objects/{class_id}/instances/{instance_id}/attributes* returns the list of attributes supported by a specific instance of a given object class. The same endpoint also works for instance 0 to retrieve the class-level attributes of the object. Each entry in the response is a CIPAttribute object containing the attribute ID, name, CIP data type, and a human-readable description. This endpoint enables a client to discover the full attribute model of any object instance without prior knowledge of the device's

implementation. A representative response for instance 1 of the Identity Object (class 1) is shown in Figure 13.


```
[
  {
    "attribute_id": 1,
    "name": "Vendor ID",
    "data_type": "UINT",
    "description": "Identification number assigned to each vendor by ODVA"
  },
  {
    "attribute_id": 2,
    "name": "Device Type",
    "data_type": "UINT",
    "description": "General type of the product"
  },
  {
    "attribute_id": 7,
    "name": "Product Name",
    "data_type": "SHORT_STRING",
    "description": "Human-readable identification of the product"
  }
]
```


**Figure 13 JSON response from GET /objects/{class_id}/instances/{instance_id}/attributes**

Once a client has identified an attribute of interest, *GET /objects/{class_id}/instances/{instance_id}/attributes/{attribute_id}/value* retrieves its current live value directly from the device. The response is a CIPAttributeValue object containing the CIP data type and the current value represented as a string. Returning the value as a string regardless of the underlying CIP data type simplifies the response model and avoids ambiguity in JSON representations of CIP-specific types such as USINT or SHORT_STRING. Clients that require typed values can use the accompanying data_type field to parse or cast the value appropriately. A representative response is shown in Figure 14.


```
{
  "data_type": "UINT",
  "value": "1"
}
```


**Figure 14 JSON response from GET /objects/{class_id}/instances/{instance_id}/attributes/{attribute_id}/value**

Both endpoints return a 404 response if the specified class ID, instance ID, or attribute ID is not found on the device, providing clear and actionable feedback to clients that query resources that do not exist. This contrasts with the 200 OK response returned on success — where 200 indicates that the resource was found and the response body contains the requested data, 404 indicates that the resource does not exist on this device. In both cases the API has functioned correctly; the status code communicates the outcome of the request.

Together, these two endpoints form the foundation for programmatic live data access — enabling a client to systematically traverse the object model of a device, discover what attributes are available, and retrieve their current values, all through standard HTTP requests and without requiring a CIP stack.

 ©2026 ODVA, Inc.

Not all attribute values are simple scalar types. The LastConflictDetected attribute (instance attribute 11) of the TCP/IP Interface Object (Class 0xF5, instance 1) is a structure containing three members of different data types. To preserve type information for each member, structured values are represented as nested objects, where each member carries its own data_type and value fields, mirroring the top-level response structure. This makes the response self-describing — a client can determine the data type of every element without any prior knowledge of the object's data model. Figure 15 shows the response for this attribute.

Byte-array members such as remote_mac and arp_pdu are represented as hexadecimal-formatted strings, a common convention for MAC addresses and raw PDU data. This does not conflict with the decimal-only requirement for JSON numbers discussed earlier, since these values are encoded as strings rather than numeric literals.


```
{
  "data_type": "STRUCT",
  "value": {
    "acd_activity": {
      "data_type": "USINT",
      "value": 2
    },
    "remote_mac": {
      "data_type": "ARRAY[6] USINT",
      "value": "001D9CC83A5B"
    },
    "arp_pdu": {
      "data_type": "ARRAY[28] USINT",
      "value": "000108000604000200 1D9CC83A5B0A000101000000000000A000102"
    }
  }
}
```


**Figure 15 JSON response from GET /objects/245/instances/1/attributes/11/value**

### Device Documentation

The /documentation endpoint returns a list of documentation resources associated with the device. This provides a standardized mechanism for vendors to expose relevant product documentation — such as user manuals, installation guides, and EDS files — directly through the device API, without requiring clients to search vendor websites or consult external registries.

Each entry in the response contains a title, a human-readable description, and an availability field indicating whether the document is accessible on the device, at an external location, or both. Where a document is available on the device, a device_url field provides its absolute URL. Where a document is available at an external location, an external_url field provides the vendor-hosted URL. Both URL fields may be present when a document is available from both sources.

A representative response is shown in Figure 16.


```
[
  {
    "title": "User Manual",
```

```
    "description": "Complete user manual covering installation, configuration, and
operation",
    "availability": "both",
    "device_url": "https://192.168.86.30/docs/user-manual.pdf",
    "external_url": "https://www.vendor.com/products/1756-L84ES/user-manual.pdf"
  },
  {
    "title": "Installation Guide",
    "description": "Quick-start installation and wiring guide",
    "availability": "external",
    "external_url": "https://www.vendor.com/products/1756-L84ES/installation-
guide.pdf"
  },
  {
    "title": "EDS File",
    "description": "Electronic Data Sheet for use with CIP configuration tools",
    "availability": "device",
    "device_url": "https://192.168.86.30/docs/1756-L84ES.eds"
  }
]
```


**Figure 16 JSON response from GET /documentation**

# Example Use Cases

The following use cases illustrate how the proposed API can be applied in practical scenarios. Each example demonstrates how standardized HTTP access to device information supports activities that would otherwise require specialized tooling, manual effort, or proprietary interfaces.

## Device discovery

In a typical industrial environment, a network may contain dozens or hundreds of CIP-enabled devices from multiple vendors. Identifying what is present on the network — and understanding the basic characteristics of each device — is a fundamental prerequisite for commissioning, maintenance, and asset management workflows.

With the proposed API, a client can perform device discovery by issuing a *GET /identity* request to each reachable device on the network. The response provides an immediate and structured summary of the device — its vendor, device type, product code, revision, and product name — without requiring any CIP-specific tooling or knowledge of the device's configuration. Because the interface is based on standard HTTP, this can be accomplished using general-purpose network scanning tools, scripts, or data collection platforms that have no awareness of CIP. Figure 17 illustrates how a single curl command issued from a standard command line is sufficient to retrieve the full identity of a device — no drivers, no configuration tools, and no knowledge of CIP messaging required.

 ©2026 ODVA, Inc.

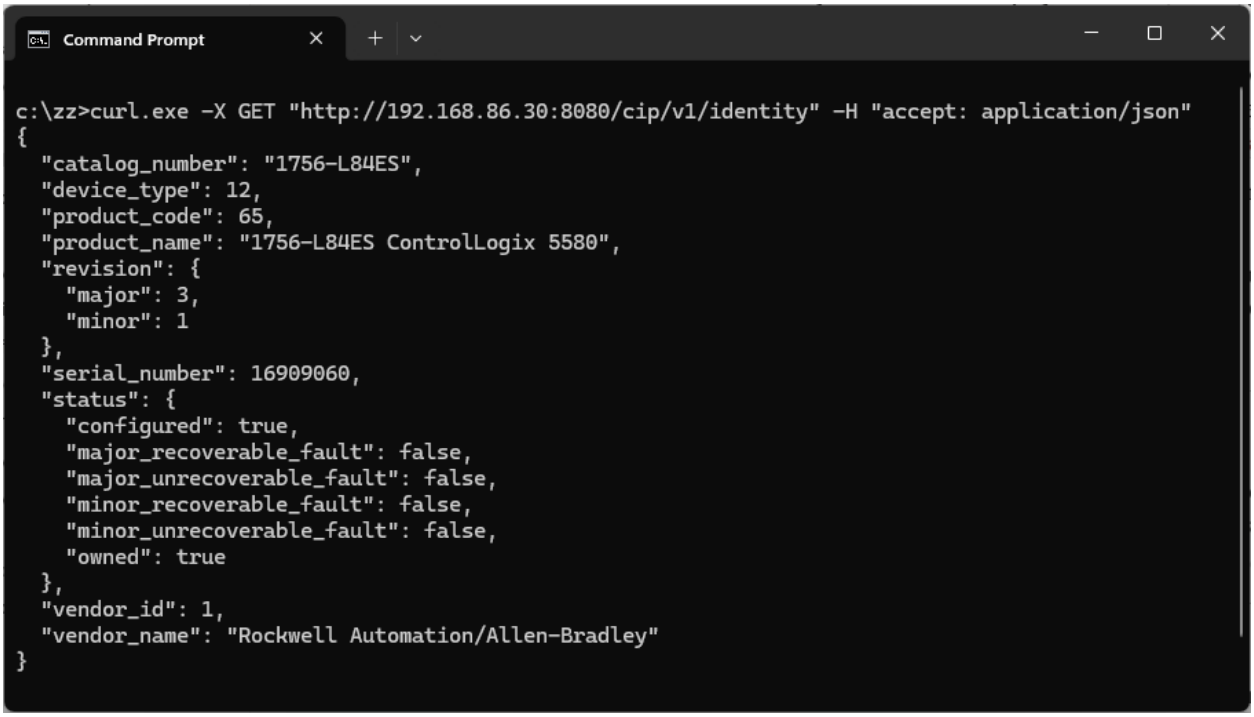



**Figure 17 Retrieving device identity using curl**

Another example is a Python script iterating over a known IP range, collecting identity responses from all reachable devices and populating an asset register automatically. Figure 18 illustrates a simple example that scans all addresses in the 192.168.86.0/24 subnet, issuing a GET /identity request to each address in turn. The structured JSON responses can be ingested directly into a database, a Computerized Maintenance Management System (CMMS), or a digital twin platform, with no parsing of EDS files or proprietary data formats required.

```python
import requests

for i in range(1, 255):
    url = f"http://192.168.86.{i}:8080/cip/v1/identity"
    try:
        response = requests.get(url, timeout=1)
        if response.status_code == 200:
            identity = response.json()
            print(f"{url}: {identity['product_name']} ({identity['vendor_name']})")
    except requests.exceptions.RequestException:
        pass
```

**Figure 18 Python script for scanning a subnet for CIP devices**

Yet another example is the use of a VBA script in Microsoft Excel to query devices and populate a workbook directly with the results. This approach requires no additional software beyond Excel itself, making it immediately accessible to engineers and system integrators who routinely work with spreadsheets for asset documentation and reporting. Figure 19 shows a VBA subroutine that accepts a device address as input, issues a GET /identity request to the corresponding device, and writes the returned attributes — such as product name, vendor, device type, and revision — into individual cells in the workbook. The result is a populated asset table that can be refreshed on demand, formatted, filtered, and shared using familiar Excel workflows. Figure 20 shows the resulting Excel output.

```
Sub GetDeviceIdentity(deviceNumber As Integer)
    Dim http As Object
    Dim json As String
    Dim ws As Worksheet

    Set ws = ThisWorkbook.Sheets(1)
    Set http = CreateObject("MSXML2.XMLHTTP")

    http.Open "GET", "http://192.168.86." & deviceNumber & ":8080/cip/v1/identity",
False
    http.setRequestHeader "Cache-Control", "no-cache"
    http.setRequestHeader "Pragma", "no-cache"
    http.Send

    If http.Status = 200 Then
        json = http.responseText

        ' Extract data and write headers in column A, values in column B
        ws.Cells(1, 1).Value = "Vendor ID"
        ws.Cells(1, 2).Value = GetJsonValue(json, "vendor_id")
    Else
        MsgBox "Error: " & http.Status
    End If
End Sub
```

**Figure 19 VBA subroutine for retrieving device identity**

 ©2026 ODVA, Inc.

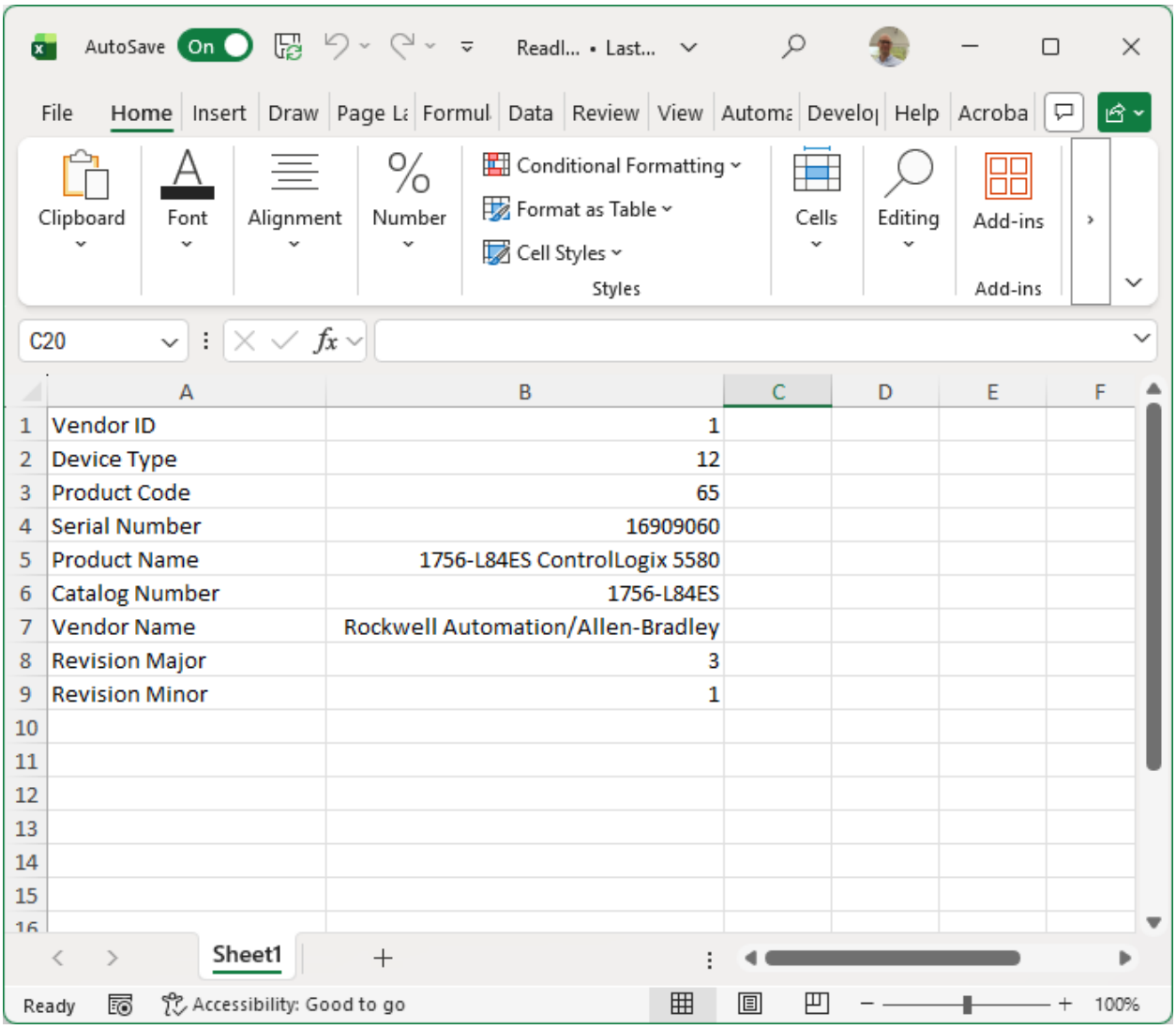
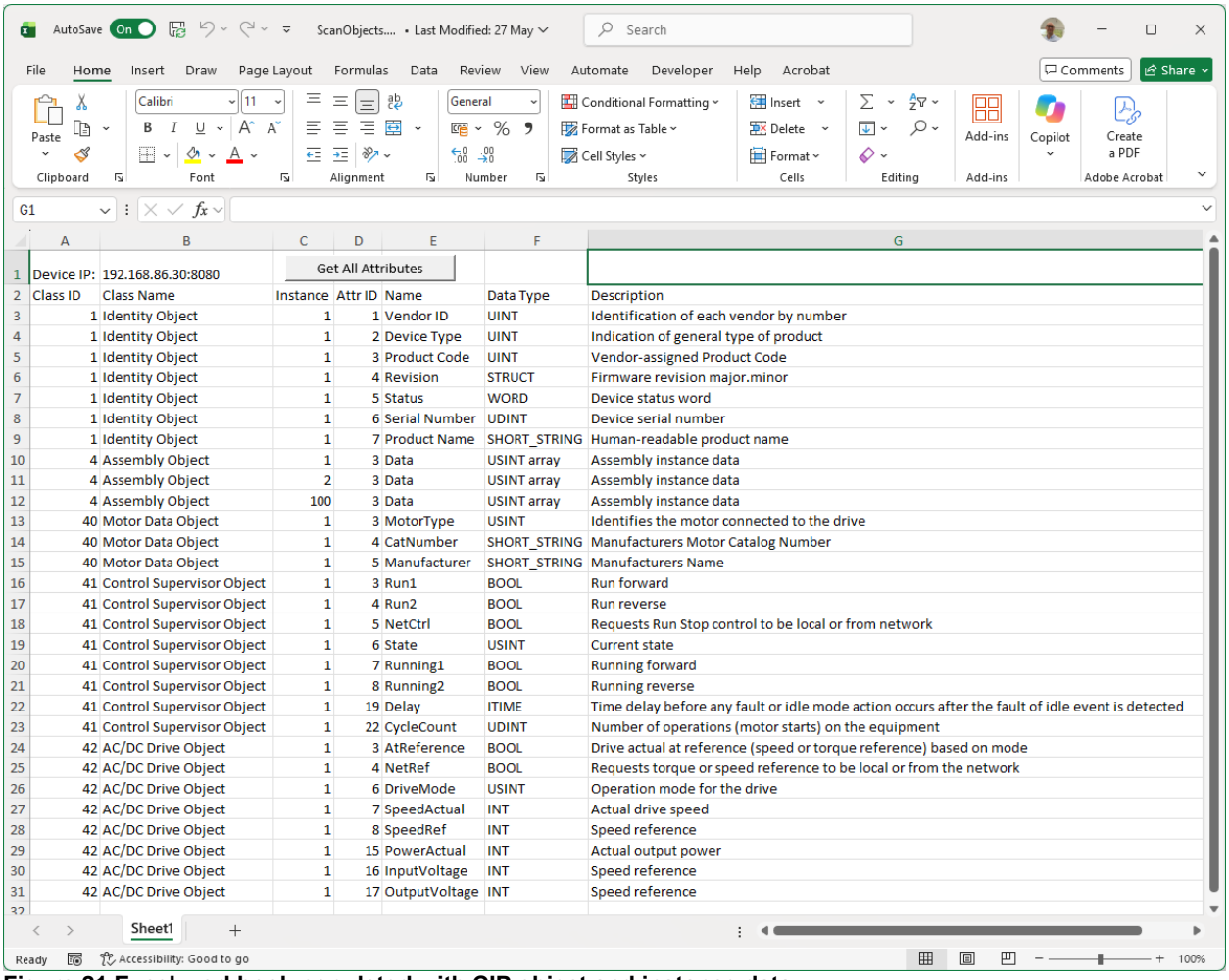



**Figure 20 Excel workbook populated with device identity data**

## Feature identification

Once a device has been identified, a common next step is to determine what it is capable of — which object classes it implements, and what instances of those objects exist. This information is essential for system integrators configuring connections, engineers developing device profiles, and tools that need to interact programmatically with the device's data model.

The *GET /objects* endpoint provides an immediate answer to the question of what a device implements, returning a concise list of all supported object classes with no need for prior knowledge of the device. Where more detail is needed, GET /objects/instances extends this with the instance numbers for each class, giving a complete picture of the full object model implemented on the device in a single request. This makes it straightforward to determine exactly what a device supports at runtime, without consulting EDS files or vendor documentation. The VBA subroutine introduced in the Device Discovery section can easily be extended to query these endpoints in the same manner, writing the returned object classes and instance numbers directly into an Excel workbook. An example of such output is shown in Figure 21.



| Device IP: | 192.168.86.30:8080 | | | Get All Attributes | | |
|---|---|---|---|---|---|---|
| Class ID | Class Name | Instance | Attr ID | Name | Data Type | Description |
| 1 | Identity Object | 1 | 1 | Vendor ID | UINT | Identification of each vendor by number |
| 1 | Identity Object | 1 | 2 | Device Type | UINT | Indication of general type of product |
| 1 | Identity Object | 1 | 3 | Product Code | UINT | Vendor-assigned Product Code |
| 1 | Identity Object | 1 | 4 | Revision | STRUCT | Firmware revision major.minor |
| 1 | Identity Object | 1 | 5 | Status | WORD | Device status word |
| 1 | Identity Object | 1 | 6 | Serial Number | UDINT | Device serial number |
| 1 | Identity Object | 1 | 7 | Product Name | SHORT_STRING | Human-readable product name |
| 4 | Assembly Object | 1 | 3 | Data | USINT array | Assembly instance data |
| 4 | Assembly Object | 2 | 3 | Data | USINT array | Assembly instance data |
| 4 | Assembly Object | 100 | 3 | Data | USINT array | Assembly instance data |
| 40 | Motor Data Object | 1 | 3 | MotorType | USINT | Identifies the motor connected to the drive |
| 40 | Motor Data Object | 1 | 4 | CatNumber | SHORT_STRING | Manufacturers Motor Catalog Number |
| 40 | Motor Data Object | 1 | 5 | Manufacturer | SHORT_STRING | Manufacturers Name |
| 41 | Control Supervisor Object | 1 | 3 | Run1 | BOOL | Run forward |
| 41 | Control Supervisor Object | 1 | 4 | Run2 | BOOL | Run reverse |
| 41 | Control Supervisor Object | 1 | 5 | NetCtrl | BOOL | Requests Run Stop control to be local or from network |
| 41 | Control Supervisor Object | 1 | 6 | State | USINT | Current state |
| 41 | Control Supervisor Object | 1 | 7 | Running1 | BOOL | Running forward |
| 41 | Control Supervisor Object | 1 | 8 | Running2 | BOOL | Running reverse |
| 41 | Control Supervisor Object | 1 | 19 | Delay | ITIME | Time delay before any fault or idle mode action occurs after the fault of idle event is detected |
| 41 | Control Supervisor Object | 1 | 22 | CycleCount | UDINT | Number of operations (motor starts) on the equipment |
| 42 | AC/DC Drive Object | 1 | 3 | AtReference | BOOL | Drive actual at reference (speed or torque reference) based on mode |
| 42 | AC/DC Drive Object | 1 | 4 | NetRef | BOOL | Requests torque or speed reference to be local or from the network |
| 42 | AC/DC Drive Object | 1 | 6 | DriveMode | USINT | Operation mode for the drive |
| 42 | AC/DC Drive Object | 1 | 7 | SpeedActual | INT | Actual drive speed |
| 42 | AC/DC Drive Object | 1 | 8 | SpeedRef | INT | Speed reference |
| 42 | AC/DC Drive Object | 1 | 15 | PowerActual | INT | Actual output power |
| 42 | AC/DC Drive Object | 1 | 16 | InputVoltage | INT | Speed reference |
| 42 | AC/DC Drive Object | 1 | 17 | OutputVoltage | INT | Speed reference |

**Figure 21 Excel workbook populated with CIP object and instance data**

This capability is particularly valuable in heterogeneous environments where devices from multiple vendors must be integrated. Rather than consulting vendor documentation or EDS files for each device, an integration tool can query the API directly and adapt its behavior based on the response — enabling a degree of runtime adaptability that is not possible with static file-based approaches.

## Large-scale data harvesting for data-driven applications

Data scientists and engineers working on predictive maintenance, process optimization, and anomaly detection require access to large volumes of device data. Collecting this data at scale — across many devices, many object instances, and many attributes — has traditionally been a significant engineering challenge, requiring deep knowledge of CIP and custom integration work for each device type.

The proposed API provides a systematic path through the device's data model that any HTTP client can follow. Starting from *GET /objects/instances*, a client can enumerate all object classes and their instances on a device. For each instance, *GET /objects/{class_id}/instances/{instance_id}/attributes* returns the full list of available attributes. Finally, *GET /objects/{class_id}/instances/{instance_id}/attributes/{attribute_id}/value* retrieves the current live value of each attribute. By traversing this hierarchy, a client can harvest the complete set of readable attribute values from a device without any prior knowledge of its implementation.

This traversal pattern is well suited to automated data collection pipelines. A data platform that supports HTTP-based data sources can be pointed at a device and configured to poll attribute values at a defined

interval, producing a time-series dataset that can feed directly into analytics and machine learning workflows. The standardized interface means the same pipeline can be reused across devices from different vendors without modification, dramatically reducing the integration effort associated with large-scale data collection.

To illustrate how straightforwardly this traversal can feed into a practical analytics workflow, consider the following example in which live attribute values from a motor drive are collected and analyzed for early signs of failure using a statistical anomaly detection approach, as shown in Figure 22.

```
import requests
import time
import statistics

BASE_URL = "http://192.168.86.30:8080/cip/v1"

ATTRIBUTES = {
    "motor_starts":     "/objects/41/instances/1/attributes/22/value",
    "output_power":     "/objects/42/instances/1/attributes/15/value",
    "input_voltage":    "/objects/42/instances/1/attributes/16/value",
    "output_voltage":   "/objects/42/instances/1/attributes/17/value",
}

WINDOW = 20      # number of samples to keep
Z_THRESHOLD = 2  # flag if more than 2 standard deviations from mean

history = {key: [] for key in ATTRIBUTES}

def get_value(path):
    response = requests.get(BASE_URL + path, timeout=2)
    if response.status_code == 200:
        return float(response.json()["value"])
    return None

def check_anomaly(key, value):
    samples = history[key]
    if len(samples) < WINDOW:
        return
    mean = statistics.mean(samples)
    stdev = statistics.stdev(samples)
    if stdev == 0:
        return
    z = abs((value - mean) / stdev)
    if z > Z_THRESHOLD:
        print(f"  WARNING: {key} = {value} (z-score: {z:.2f}) — possible anomaly")

while True:
    print("--- Polling device ---")
```

 ©2026 ODVA, Inc.

```
    for key, path in ATTRIBUTES.items():
        value = get_value(path)
        if value is not None:
            check_anomaly(key, value)
            history[key].append(value)
            if len(history[key]) > WINDOW:
                history[key].pop(0)
            print(f"  {key}: {value}")
    time.sleep(5)
```

**Figure 22 Python script for predictive maintenance using live CIP device data**

The script polls four attributes from the device every five seconds — the number of motor starts, actual output power, actual input voltage, and actual output voltage. For each attribute, it maintains a rolling window of the twenty most recent samples. Once the window is full, each new reading is evaluated using a z-score, which measures how far the current value deviates from the recent historical mean expressed in units of standard deviation. If a value deviates by more than two standard deviations from the mean, the script prints a warning to the console identifying the attribute and the magnitude of the deviation. A sudden drop in output voltage, an unexpected spike in output power, or an unusually high number of motor starts in a short period would all trigger such a warning, giving an operator or automated system early notice of a potentially developing fault.

While intentionally simple, this example demonstrates the core principle: the standardized API makes the device's live data directly accessible to any Python-based analytics workflow, with no CIP-specific code required beyond the HTTP requests themselves. The same pattern can be extended to collect data from multiple devices simultaneously, feed values into a time-series database, or integrate with more sophisticated machine learning models — all using standard tools and libraries that are widely available in the Python ecosystem.

## Integration with engineering tools and digital repositories

Engineering tools — including configuration software, asset management platforms, and digital repositories — increasingly need to consume structured device information as part of automated workflows. The proposed API provides a natural integration point for such tools, exposing device information through a standard interface that requires no vendor-specific plugins or adapters.

A digital repository maintaining records of deployed devices can use the API to keep its records current by periodically querying *GET /identity* for each device and updating its records when changes in revision, product name, or status are detected. An engineering tool generating device documentation can use *GET /objects/instances* and the attribute endpoints to build a complete description of the device's data model programmatically, rather than relying on manually maintained EDS files. A commissioning tool validating a newly installed device can compare the identity and object inventory returned by the API against an expected profile, flagging discrepancies automatically.

In each of these scenarios, the value of the proposed API lies not only in the information it exposes, but in the consistency and accessibility of the interface through which that information is delivered. Because the API conforms to the OpenAPI Specification, engineering tools can generate client code directly from the definition, reducing integration effort and ensuring that the tool's behavior remains aligned with the API contract as it evolves.

To further illustrate the breadth of integration possibilities, a Power Automate flow was developed that periodically queries a CIP device, parses the JSON responses, and formats the retrieved data into a structured dataset that is automatically refreshed in Power BI. The flow requires no custom code or CIP-specific tooling — it uses standard HTTP actions available natively in Power Automate to issue requests

to the device API, extracts the relevant fields from the JSON responses, and passes the structured data directly to Power BI. While this example targets a single device, the same flow can readily be extended to poll multiple devices in sequence, aggregating their data into a unified dataset. Figure 23 shows the Power Automate flow. Once the data is available in Power BI, it can be visualized in interactive dashboards, trended over time, filtered by device or attribute, and shared across an organization without any additional infrastructure. Alerts and thresholds can be configured to notify stakeholders when values fall outside acceptable ranges, and the reports can be accessed from any device with a web browser — bringing operational device data directly into the business intelligence tools that engineers and managers already use. Figure 23 shows an example Power BI report refreshed from the device API.

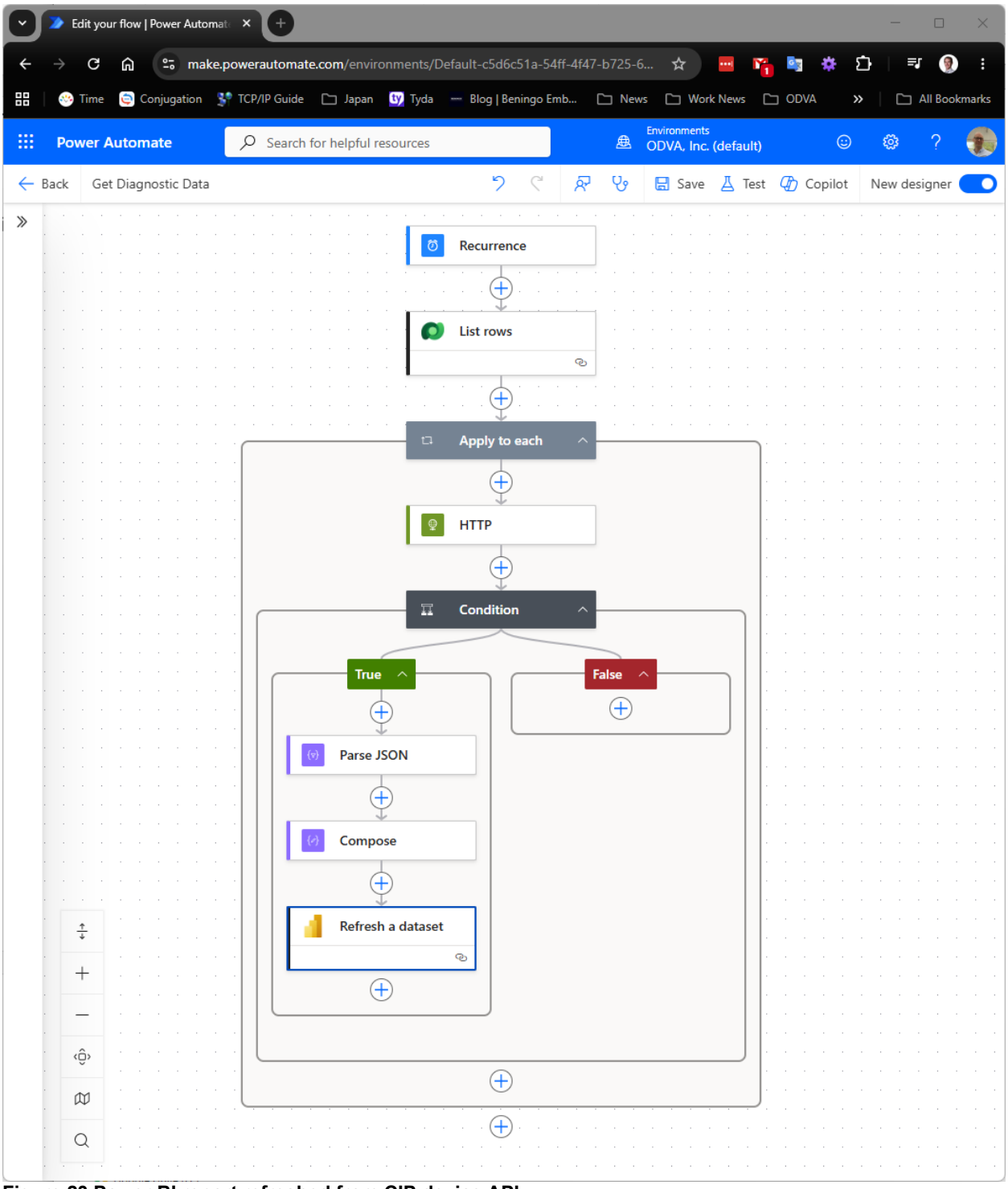



**Figure 23 Power BI report refreshed from CIP device API**

## Accessing device documentation

A common need for maintenance engineers and system integrators is quick access to the documentation for a specific device — the user manual, wiring guide, or EDS file — without having to search vendor websites or consult internal document repositories.

With the proposed API, retrieving the documentation available for a device requires nothing more than a web browser. Navigating to the device's /documentation endpoint returns a structured list of all documentation resources the device exposes, including titles, descriptions, and direct download links. Because the response is standard JSON, any modern browser renders it immediately and the URLs in the response are directly clickable.

For devices that also host documentation locally — such as an EDS file or a product manual stored in device — the device_url field provides a direct link to the file served from the device itself. For externally hosted documents, the external_url field points to the vendor's website. In either case, the engineer needs no knowledge of where the documentation is stored or how it is organized — the device itself provides the answer.

This represents perhaps the most accessible demonstration of the proposed API's value: a technician standing next to a piece of equipment can retrieve its full documentation index using the browser on a laptop or mobile phone, with no installed software, no credentials, and no prior knowledge of the device beyond its IP address.

A representative response is shown in Figure 24.

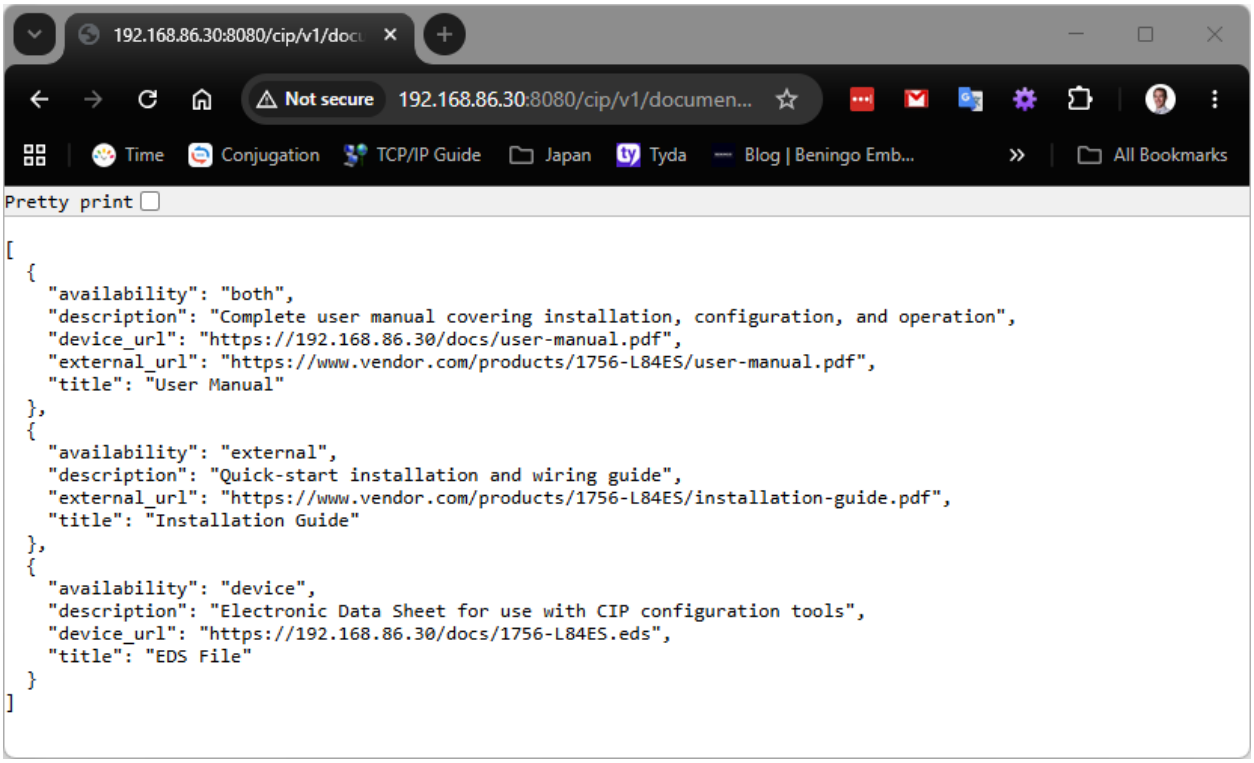



**Figure 24 GET /documentation response rendered in a web browser**

# Discussion

## Benefits for system integrators, engineers, application developers, and data scientists

The proposed API addresses the needs of a broad range of stakeholders, each of whom interacts with CIP-enabled devices in fundamentally different ways but share a common need for structured, accessible device information.

For system integrators, the API reduces the time and effort required to inventory and characterize devices on a network. Rather than relying on manual inspection, proprietary configuration tools, or EDS file parsing, an integrator can query any compliant device over standard HTTP and receive a structured, immediately usable response. The device discovery and feature identification use cases presented in this paper illustrate how tasks that would previously have required specialized tooling can be accomplished with a simple script or even a single command.

For business controllers and managers, the API provides a reliable and consistent source of device information that can be integrated directly into the tools and platforms they already use for reporting and decision-making. The examples presented in this paper — ranging from Excel workbooks populated via VBA to Power BI reports refreshed through Power Automate — demonstrate that once the integration is in place, operational device data becomes accessible without requiring any knowledge of CIP or industrial networking. While the initial setup may require assistance from engineers or IT personnel to configure the API calls and connect the data sources, the day-to-day consumption of the data requires no technical expertise beyond familiarity with standard business tools.

For application developers, the standardized interface significantly lowers the barrier to building tools and applications that work across devices from multiple vendors. Because the API conforms to the OpenAPI Specification, client code can be generated automatically from the definition, and the behavior of the interface is fully specified without reliance on vendor-specific documentation. A developer building an asset management platform, a commissioning tool, or a data collection pipeline can target the API directly, confident that any compliant device will respond in the expected way.

For data scientists, the API provides a systematic and programmatic path to live device data that requires no CIP expertise. The traversal pattern described in the large-scale data harvesting use case — enumerating objects, discovering attributes, and polling values — makes it possible to collect structured time-series data from any compliant device using standard HTTP tooling and feed it directly into analytics workflows, machine learning pipelines, or visualization platforms.

## Limitations and open challenges

While the proposed API offers significant benefits, a number of limitations and open challenges should be acknowledged.

The current definition covers device identity, object discovery, and attribute value retrieval. It does not address write access to device attributes, which would be necessary for configuration and control use cases. Extending the API to support write operations introduces additional complexity, particularly around authorization and the risk of unintended modification of device state, and is left for future work.

The API is hosted on the device itself, which requires the device to have sufficient processing and networking resources to serve HTTP requests alongside its primary communication functions. While this is increasingly feasible given the capabilities of modern embedded systems, it may present challenges for resource-constrained devices or legacy hardware. In such cases, a gateway or proxy architecture — in which a separate device hosts the API on behalf of one or more CIP devices — may be a more practical deployment model.

The device-hosted model also has implications for network architecture. Because the API requires an inbound HTTP connection to each device, reaching devices from an external network — such as a CMMS on the IT side of an OT/IT boundary — requires a firewall rule permitting inbound access to every device individually. This contrasts with protocols such as MQTT, where devices initiate outbound connections to a broker; while this avoids the need for inbound firewall rules, it also means the network boundary no longer restricts where that data can be sent, since the broker connection is typically permitted for outbound traffic in general rather than scoped to a specific known destination. The proposed API's inbound model, though it requires more firewall configuration, allows more precise control over exactly which external hosts may reach which devices. This is a genuine trade-off of the device-hosted

model in networks with a hard OT/IT boundary, and is discussed further as an open question for future work.

Security is an important consideration that the current proposal addresses only at a high level. The use of HTTPS provides transport-level encryption and authentication, but a complete security model would need to address authorization — controlling which clients are permitted to read which attributes — as well as protection against denial-of-service attacks and other threats specific to the industrial environment. The relationship between the proposed API and CIP Security deserves further study.

Finally, the practical adoption of the proposed API depends on vendor support. The value of a standardized interface is realized only when a sufficient number of devices implement it consistently. Establishing the definition as a recognized standard — whether through ODVA or another body — and providing reference implementations and conformance test tools would be important steps toward broad adoption.

## Comparison with alternative approaches

The proposed API is not the first attempt to address the need for structured, accessible device information in industrial automation, and it is worth considering how it relates to existing approaches.

EDS files remain the most widely used mechanism for describing CIP device capabilities. They are well understood, broadly supported, and deeply integrated into existing tooling ecosystems. However, EDS files are static artifacts and are not designed for programmatic access over a network and require EDS-specific tooling to parse and interpret. The proposed API complements rather than replaces EDS files — it provides a web-friendly, dynamically queryable interface to the same underlying device information, making it accessible to a much wider range of clients and tools.

The Extended Logical segment mechanism proposed in prior work at this conference addresses a related but distinct problem: making device metadata discoverable online through existing CIP infrastructure. The proposed API takes a different approach, providing a REST interface that is independent of CIP messaging and accessible to any HTTP client. The two approaches are complementary, and a device could implement both without conflict.

i3X, the open REST API specification coordinated by CESMII, addresses the interoperability of manufacturing data at the platform level. While i3X and the proposed API share a common foundation in REST and OpenAPI, they operate at different levels of abstraction — i3X provides a unified interface for accessing contextualized manufacturing data across platforms, while the proposed API focuses specifically on the identity, object model, and live attribute data of individual CIP devices. The two specifications could coexist and potentially interoperate, with i3X consuming data surfaced by the proposed API as part of a broader data integration architecture.

OPC UA addresses a similar need for structured, machine-readable device information, and — like the proposed API — has moved beyond its traditional binary transport to support web-friendly encodings, including a JSON representation and WebSocket-based transport. The key distinction is not protocol accessibility but client capability: consuming OPC UA requires an OPC UA client stack or SDK, along with familiarity with its address-space and node-hierarchy information model. HTTP, by contrast, is a capability that IT-side tools already have built in. Every example in this paper — curl, an Excel VBA macro, a Power Automate flow — relies on nothing more than the ability to issue an HTTP request, a capability present by default in virtually every modern programming language, scripting environment, and business application. None of these tools has native OPC UA support, and it is unlikely that mainstream applications such as Excel or Power Automate ever will; OPC UA client functionality is not a feature these tools are built around, and adding it would require a dedicated SDK integration that most such applications have no reason to invest in. This asymmetry is the practical argument for the proposed API: it targets exactly the class of clients that already speaks HTTP but will, in practice, never speak OPC UA. OPC UA also provides a considerably more mature built-in security and authorization model than the high-level HTTPS-only approach currently proposed here (see "Limitations and open challenges"). The proposed API is not

intended as a replacement for OPC UA's broader modeling and security capabilities; rather, it targets a narrower goal — minimal-friction access to a well-bounded set of CIP device attributes for clients that do not already have, and are unlikely to ever gain, an OPC UA stack.

MQTT differs from the proposed API more fundamentally: it is a publish/subscribe transport protocol, not a data modeling framework. MQTT itself defines nothing about payload structure, device identity, or attribute semantics — a message is an opaque byte string from the protocol's perspective, and any structure, schema, or discoverability has to be defined entirely by the application built on top of it. This stands in contrast to the proposed API, OPC UA, and i3X, each of which defines a standardized, machine-readable device information model as part of the specification itself. MQTT's relevant strength lies elsewhere: clients, including devices publishing data, typically initiate outbound connections to a broker, which is often easier to permit through firewalls separating OT and IT networks than an inbound connection to a server hosted on the device — the model the proposed API currently requires (see "Limitations and open challenges"). The two approaches are not direct substitutes: MQTT is well suited to continuous, event-driven telemetry once a payload structure has been agreed upon out of band, while the proposed API provides that structure — and the means to discover it — without requiring any prior agreement between publisher and consumer. A device could reasonably expose both, using MQTT for streaming telemetry with an application-defined payload and the proposed API for self-describing discovery and identity information.

# Conclusion

## Summary of contributions

This paper has presented a proposed API based on the OpenAPI Specification for describing and accessing information from CIP-enabled devices. The proposed definition provides a standardized REST interface through which key device information — including identity attributes, implemented object classes and instances, and live attribute values — can be retrieved in a consistent and machine-readable form by any HTTP-capable client, without requiring specialized CIP tooling or expertise.

The paper has demonstrated through a series of practical use cases that the proposed API is broadly accessible and immediately applicable. Device discovery can be performed using a single curl command or a short Python script. Asset inventories can be populated automatically into Excel workbooks using VBA. Operational device data can be surfaced in Power BI dashboards through Power Automate flows, making it available to business controllers and managers without any knowledge of the underlying industrial protocol. Live attribute data can be collected and analyzed for predictive maintenance using standard Python libraries. In each case, the same standardized interface serves clients ranging from command-line tools to enterprise business intelligence platforms, across devices from any vendor that implements the definition.

The proposed API complements existing CIP mechanisms — including EDS files and the Extended Logical segment — rather than replacing them, and is grounded in the OpenAPI Specification, which brings with it a rich ecosystem of tooling for code generation, documentation, and validation. The definition is versioned, read-only, and designed with simplicity and extensibility as core principles.

## Future work

Several directions for future work have been identified. Write access to device attributes would extend the API from a read-only information interface to a full configuration and control interface, enabling a broader range of integration scenarios. A complete security model — addressing authorization, access control, and the relationship with CIP Security — would be a prerequisite for write access and an important consideration for production deployments in sensitive environments.

The current definition covers the core information model of a CIP device. Future extensions could address additional aspects of the device's capabilities, including support for EDS file retrieval and richer

metadata aligned with the proposals from prior work at this conference. A gateway or proxy architecture for resource-constrained or legacy devices that cannot host the API directly is also worth exploring.

Finally, and most importantly, adoption depends on end-user value: vendors will only invest in implementing this API if their customers ask for it and see clear benefit in it. The value of a standardized interface is therefore realized in two steps — end users must recognize the accessibility, automation, and interoperability benefits it enables, and vendors must, in turn, see sufficient demand to justify implementing it in their devices. Establishing the proposed definition as a recognized standard, developing reference implementations, and providing conformance test tools would give vendors a clear and supported path to implementation and give the broader community confidence that any compliant device will behave as expected. This paper represents a first step in that direction — an invitation to the CIP community to engage with the proposed approach and contribute to its further development.

## References


[1] Ecma International, "ECMA-404: The JSON Data Interchange Syntax," 2nd edition, December 2017. https://ecma-international.org/publications-and-standards/standards/ecma-404/

[2] OpenAPI Initiative, "OpenAPI Specification v3.1.0," 2021. https://spec.openapis.org/oas/v3.1.0.html

[3] G. Majcher, "Enabling Data Scientist Use Cases with Discoverability and Metadata," ODVA Industry Conference, 2023. https://www.odva.org/library_proceedings/enabling-data-scientist-use-cases-with-discoverability-and-metadata/

[4] G. Majcher and M. Schaffner, "Metadata for CIP Devices," ODVA Industry Conference, 2025. https://www.odva.org/library_proceedings/metadata-for-cip-devices/

[5] CESMII, "i3X — Industrial Information Interoperability eXchange," open API specification, available at https://www.i3x.dev/

[6] ODVA, The CIP Networks Library, Volume 8, CIP Security, Ann Arbor: ODVA, Inc., 2015-2026.

[7] O. Ben-Kiki, C. Evans, and I. döt Net, "YAML Ain't Markup Language (YAML) Version 1.2," 3rd edition, October 2009, revised 2021. https://yaml.org/spec/1.2.2/

[8] ODVA, The CIP Networks Library, Volume 1: Common Industrial Protocol, Ann Arbor: ODVA, Inc., 2001-2026.

[9] SmartBear Software, "Swagger — API Development for Everyone," available at https://swagger.io



**********

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.