

The Arrowhead Framework Ontology

OSKAR WINTERCORN  AND JAN VAN DEVENTER 

Department of Computer Science, Electrical and Space Engineering, Luleå University of Technology, 971 87 Luleå, Sweden

CORRESPONDING AUTHOR: OSKAR WINTERCORN (e-mail: oskar.wintercorn@ltu.se).

This work was supported by the European Union through the Chips Joint Undertaking Arrowhead fPVN project under Grant 101111977.

ABSTRACT Modern industrial applications are increasingly built from independently developed systems that discover each other at run time and compose into mission-specific behavior. The challenge is not only to collect data, but to preserve enough context to understand and diagnose the behavior of the resulting system-of-systems. This article presents the arrowhead framework ontology, an explicit, queryable semantic representation of Arrowhead framework deployments. Using a climate-control demonstrator, we export timestamped resource description framework (RDF) snapshots from live systems, ingest them into GraphDB, validate them with shapes constraint language, and apply web ontology language reasoning to derive additional facts about structure and dependencies. We show that this enables practical diagnosis and impact analysis via SPARQL protocol and RDF query language, even though none of the individual systems were designed to answer such questions. The same representation also supports digital threads for traceability and lifecycle insight. Together, these results illustrate how explicit semantics can provide a nonintrusive path to more transparent and evolvable industrial systems.

INDEX TERMS Arrowhead framework, digital thread, GraphDB, knowledge graphs, semantic web, SHACL, system-of-systems (SOS), web ontology language (OWL 2).

I. INTRODUCTION

Many argue that information is paramount. Consistent with this trend, global data creation is commonly estimated at 0.4–0.5 ZB per day, depending on definitions and measurement methodology [1]. Yet information without meaning is of little value, a fact that becomes clear when you try to read a newspaper in a language you do not understand. Knowledge arises when information is interpreted and given meaning. Intelligence is the ability to apply that knowledge to make sound decisions across the different domains of life. Digital transformation, in turn, is about converting vast stores of digital information into actionable knowledge that improves decision-making across operations, products, and strategy.

Two pivotal terms in the previous paragraph—meaning and intelligence—warrant a brief pause. Semantics is the study of meaning, and this naturally suggests the Semantic Web as a way to attach machine-interpretable meaning to digitally stored information [2]. Defining intelligence is more contentious. Legg and Hutter argue for a notion of intelligence that spans very different systems, from humans and other animals, such as dogs, to artificial agents and learning algorithms [3]. This raises a useful thought experiment:

which is more intelligent, a dog or a thermostat? In a narrow domain, such as climate control, a thermostat may outperform a dog. Yet neither system, on its own, can explain that an unusually low indoor temperature might be caused not only by cold weather but by a faulty sensor or a malfunctioning heater. This illustrates the limitation of narrow, task-specific behavior: robust interpretation and diagnosis require explicit contextual knowledge to support automated reasoning.

The goal of this article is to show that digital transformation can be achieved without replacing existing business infrastructure, that is, without changing the underlying set of assets built on diverse technologies. Instead, by enriching and connecting existing data with explicit semantics, it becomes possible to infer new knowledge, detect anomalies, and create end-to-end digital threads, including plant history, product provenance [for example, digital product passports (DPP)], and environmental metrics, such as carbon footprint [life cycle assessment (LCA)]. The same factual backbone can also be used to ground large language models, reducing hallucinations by anchoring generated outputs in verifiable data [4].

In industrial production systems, the central challenge is not only data volume but preserving context across layers,

systems, and organizational boundaries. Industrial digitization accelerated with Industry 3.0, as computers and automated control systems began producing large volumes of operational data. Because connectivity was limited, this data typically remained local to plants and organizational silos, and information flows largely followed the enterprise–control system integration hierarchy defined by ISA-95 (IEC 62264) [5]. As networking matured and operational technology (OT) and information technology (IT) environments became interconnected via IP-based infrastructure and the Internet, data could move across edge, plant, and cloud boundaries. This connectivity is a prerequisite for Industry 4.0. Reference architectures, such as RAMI 4.0, formalize this edge-to-cloud perspective [6]. Industry 4.0 builds on cyber-physical systems and the Internet of Things (IoT), where assets sense, compute, and act while exposing software interfaces for coordination. At the same time, decentralization increases agility and scalability but also intensifies diversity in protocols, data models, and semantics. The result is that end-to-end interpretation and diagnosis become harder, especially in complex production systems [7], [8], [9].

To move from information to knowledge in such heterogeneous environments, systems must make semantics explicit rather than implicit in application code or interface documentation. A common approach is to use Semantic Web technologies, in particular the Web Ontology Language (OWL 2), to represent domain knowledge in a machine-interpretable form [10]. In OWL 2, knowledge is represented using classes, properties, and individuals, together with axioms that formalize their meaning and enable logical inference over their relationships. An ontology provides the shared conceptualization of a domain by formally defining its concepts and entities, and by specifying the relations between them. Once data are linked to these concepts, automated reasoning can infer additional facts, detect inconsistencies, and support diagnosis and traceability across the edge-to-cloud continuum.

The *arrowhead framework ontology* (hereafter the AFO or the ontology), introduced here, captures the structure and relationships of Arrowhead framework local clouds, systems, and services in a formal, semantic model that is stored and queried in a triple store. By making the implicit architecture explicit, the ontology facilitates reasoning about dependencies, validates service bindings, and supports analysis in complex edge-to-cloud deployments. While it does not guarantee semantic interoperability on its own, it enhances the diagnosability of deployments, which is vital for understanding their functionality. By reducing reliance on manual interpretation, it can shorten validation cycles. To corroborate these claims, we pose the following research questions.

- Q1) Can the system-of-systems (SoS) use automated reasoning to infer new knowledge about its systems?
- Q2) Can the SoS validate incoming information while ensuring data integrity?
- Q3) Can the SoS use the AFO to enable proper diagnosis?
- Q4) Can this approach be deployed on edge devices?
- Q5) Can the SoS provide relevant digital threads and temporal traces of deployment history, illustrated here

through functional-location history and potentially extensible to DPP, LCA, and life-cycle costing (LCC)?

In addition to these research questions, we define a set of ontology-level competency questions that capture the operational information needs the AFO is intended to support, such as identifying service providers and consumers, detecting unsatisfied dependencies, locating deployment context, and reconstructing temporal traces. These competency questions are introduced in Section III and evaluated systematically in Section IV.

The rest of this article is organized as follows. Section II presents the current state of the field and provides essential background on relevant technologies. Section III introduces the AFO. Section IV showcases the ontology through a use case, highlighting the benefits of having a semantic layer in a SOS application. Section V analyzes and reflects on the research findings. Finally, Section VI concludes this article with remarks on the findings and outlines future research directions.

II. RELATED WORK

Developing and maintaining an ontology is both time-consuming and costly. It therefore requires clear goals and a good understanding of both the intended context and reusable existing ontologies. In industrial automation, practitioners must reconcile standardization efforts made, vendor-specific information models, and ad hoc integrations. Adding a semantic layer to the complexity of SoS is only truly justified if it helps manage the complexity and reduce integration efforts.

As industry moves toward more dynamic, service-oriented SoS, interoperability challenges become more acute due to late binding, heterogeneous protocols, and independently evolving stakeholders. In a practitioner-grounded survey, Sadeghi et al. [11] described interoperability as a significant challenge in large, distributed and collaborative SoS, and structured the problem across data, service, and system interoperability concerns. Their proposed architecture emphasizes a semantic approach to federated interoperability, combining registry capabilities with semantic interoperability services, such as converters, mapping mechanisms, and ontology management, to support interpretation and transformation across heterogeneous systems. Schlemitz and Mezhyuev [12] surveyed approaches for data collection and process standardization in smart manufacturing and argue that interoperability remains difficult because legacy equipment must be integrated alongside industrial-Internet-of-Things-enabled (IIoT) systems. Therefore the authors examined both communication-level standardization open platform communications unified architecture (OPC UA) and higher-level data modeling approaches, such as the asset administration shell (AAS). Taken together, these studies point to the need for shared semantics that can be reused across organizations and technology stacks rather than reinvented project by project.

One route toward such shared semantics is the adoption of reference and upper ontologies that can be reused across vertical and horizontal domains. In the smart-appliance and smart-home domain, for example, the Smart Appliance

REfERENCE ontology (SAREF) provides a shared vocabulary for devices, functions, and services, with the explicit goal of enabling semantic interoperability [13]. Case studies demonstrate how heterogeneous smart-home datasets can be mapped to SAREF and how shared SAREF-based knowledge can support concrete interoperability scenarios [14], [15]. For cross-domain sensing and actuation, the semantic sensor network ontology (SSN) and its lightweight sensor, observation, sample, and actuator (SOSA) core model sensors, observations, and features of interest [16], [17]. At the manufacturing level, the Industrial Ontologies Foundry (IOF) uses a layered ontology architecture grounded in the basic formal ontology (BFO). An IOF upper ontology descending from BFO has been proposed, and the IOF Core ontology serves as a mid-level foundation that can be specialized into domain-specific manufacturing ontologies [18], [19].

These shared ontologies can provide a stable semantic backbone, but they deliberately abstract away from platform-specific architectures and runtime execution models. Therefore, they do not directly represent the semantics of industrial automation frameworks, such as OPC UA, FIWARE, or the arrowhead framework, in particular the concepts that govern communication, discovery, orchestration, and service binding at runtime. For a practitioner, understanding the semantic state of the art around these frameworks is essential.

A. OPC UA

OPC UA is widely used throughout industry because it combines a communication protocol with a flexible information-modeling framework. Information models are commonly distributed as OPC UA NodeSet XML files, which define typed nodes and references for representing devices, assets, and processes in a platform- and application-agnostic way [7]. However, NodeSets are not natively expressed in resource description framework (RDF) or OWL, which limits direct reuse of Semantic Web tooling. Schiekofner et al. [20] provided a formal translation from OPC UA information models to OWL and demonstrate how the result can be queried via SPARQL protocol and RDF query language (SPARQL) and checked against selected modeling rules using standard Semantic Web technologies. Perzylo et al. [21] generated OWL ontologies from complete NodeSet specifications and use them as a semantic foundation for digital twins of manufacturing resources. Together, these approaches show how the structure and meaning captured in OPC UA information models can be lifted into OWL, enabling standard semantic web querying and reasoning. More recent standardization work complements these OWL lifts of OPC UA information models. In particular, the world wide web consortium Web of Things (W3C WoT) Web of Things (WoT) ecosystem uses bindings to describe how protocol-specific interfaces are exposed through thing descriptions, and the OPC Foundation has released an official OPC UA Binding for WoT. This development strengthens the bridge between OPC UA and semantic-web-compatible metadata, although it does not by itself provide the Arrowhead-specific runtime governance and dependency semantics targeted by AFO [22], [23].

In our context, OPC UA mainly addresses how to represent industrial assets and their interfaces. It therefore serves as an operational-technology data source whose information models can be lifted into an ontological representation and integrated with higher level context and service models.

B. ASSET ADMINISTRATION SHELL

Complementary work enriches industrial service descriptions (SD) and Industry 4.0 artifacts with explicit semantics. SA-OPC-UA extends SAWSDL-style annotations to OPC UA application-specific methods to assist automated method discovery and to provide a basis for subsequent method composition and orchestration [24]. SemOPC-UA similarly introduces semantic descriptions for OPC UA methods by adapting OWL-S concepts, enabling more flexible orchestration in manufacturing scenarios [25].

Beyond method descriptions, the AAS provides a standardized structure for representing asset metadata and digital representations. Bader and Maleshkova [26] proposed the semantic asset administration shell (SAAS), an RDF/OWL formalization of the AAS data model together with an XML-to-RDF mapping, selected reasoning rules, and a validation model encoded as shapes constraint language (SHACL) shapes. They evaluate mapping and data overhead and argue that the semantic lifting remains manageable; SHACL-based constraint checking can be performed during development/deployment or on more capable infrastructure [26], [27]. Leveraging the capabilities of open-world inference and reasoning together with SHACL constraints for closed-world validation aligns with the approach adopted by the AFO.

C. FIWARE

FIWARE is an open-source platform that provides reusable building blocks for smart applications, with NGSI-based context management via a Context Broker as a central component [8]. At the standardization level, ETSI next-generation-service-interface-for-linked-data-based (NGSI-LD) defines both an information model (entities described through properties and relationships) and a representational state transfer application programming interface (REST API) for providing, consuming, and subscribing to such context information; payloads are encoded in JavaScript object notation for linked data (JSON-LD) (with an RDF interpretation) [28]. Araujo et al. [29] benchmarked FIWARE in a cloud testbed designed to emulate large-scale smart-city IoT workloads and analyze scalability bottlenecks and deployment strategies. Recent smart-city work further combines NGSI-LD brokers with enrichment and linkage pipelines to harmonize heterogeneous datasets and make contextual relations explicit [30], [31].

On top of NGSI-LD, shared domain templates such as the Smart Data Models initiative provide reusable JSON-LD contexts and schemas that support consistent entity representations across projects, enabling straightforward mapping of heterogeneous sources to NGSI-LD entities in applied pipelines [30], [32]. Implementations, such as City Data Hub illustrate how NGSI-LD APIs, common data models, and marketplace components can be combined into a modular

platform, and they further provide an RDF/SPARQL view and ontology-based annotation when required [31].

From an industrial-automation perspective, FIWARE thus primarily contributes a semantically enabled layer for *entity-centric context information*. However, it does not provide a semantic model of an Arrowhead framework-style service-oriented SoS operation within local clouds, such as provider/consumer roles, orchestration decisions, and governance. This motivates an Arrowhead framework-specific ontology that complements FIWARE by formalizing service roles, dependencies, and governance.

D. ARROWHEAD FRAMEWORK

The arrowhead framework is not intended as an alternative to OPC UA, FIWARE, or other middleware technologies. Instead, it complements them by providing a structured approach for organizing, discovering, and governing industrial services within *local clouds*. In the arrowhead framework, asset functionality is exposed as services. Physical assets (sensors and actuators) and software components (control assets and databases) are encapsulated behind a software interface; the resulting unit is referred to as a *system*. A SoS for a given mission is composed by selecting and connecting appropriate systems, including arrowhead framework core systems, such as the service registry and the orchestrator. In a typical deployment, service-providing systems register their offered services with the service registry, while service-consuming systems obtain suitable providers via the orchestrator based on functional and nonfunctional constraints [33].

Both OPC UA and NGSI-LD illustrate a practical prerequisite for semantic integration in industry. The underlying models must be sufficiently structured to support automated processing and alignment. A model-based arrowhead framework implementation in Go (mbaigo) applies model-based systems engineering to capture the deployments as explicit models and to generate configuration and code artifacts from them [34]. While initially motivated by knowledge transfer to students, technicians, and engineers, these explicit deployment models also provide a direct basis for machine-interpretable knowledge. In mbaigo, the KGrapher system extracts the deployed systems and services and materializes them as a knowledge graph in a triple store (GraphDB). When this deployment graph is enriched with the AFO, additional facts about services, systems, and their relationships are inferred. Moreover, the graph links operational-technology interfaces to Arrowhead framework service abstractions: for example, OPC UA or Modbus endpoints are represented as service-providing systems, and message queuing telemetry transport (MQTT) topics exposed by brokers are related to the corresponding services. This enables integrated reasoning across device-level protocols and higher level service context.

The next section presents the design and implementation of the AFO and explains how it is aligned with these deployment models and knowledge-graph representations.

E. GAP ANALYSIS AND POSITIONING

Arrowhead framework-specific semantic resources already exist, and we therefore do not claim that the AFO is the first ontology to formalize Arrowhead concepts. For example, I40GO includes dedicated ArrowHead System and ArrowHead Service modules within a broader Industry 4.0 ontology stack, while the digital reference arrowhead ontology includes classes, such as AH Local Cloud, AH System, and AH Service, together with core-system concepts and relations, such as *produces* and *runs on* [35], [36]. These works show that key Arrowhead concepts have already been captured at the ontology level.

What distinguishes the AFO is therefore not the introduction of arrowhead framework terminology itself, but the modeling objective. In their published form, existing Arrowhead framework ontologies place stronger emphasis on conceptual coverage and, in the case of Digital Reference, on documentation and design-description artifacts, such as interface design descriptions, SD, system descriptions, system design descriptions, web application description language (WADL) artifacts, and documentation relations, such as *is documented by* and *refers to*. They are therefore valuable as conceptual and documentation-alignment resources, but they are not primarily presented as ontologies for representing the current deployment state of a local cloud.

The AFO is positioned at that operational layer. It formalizes the deployment structure and service-dependency relations needed to describe how a local cloud currently functions, including local clouds, systems, husks, unit assets, hosts, servers, provided services, consumed services, and the constraints that connect them. When instantiated from running deployments and combined with the KGrapher, timestamped snapshots, SHACL validation, OWL reasoning, and SPARQL querying, the AFO supports validation, inference, diagnosis, and traceability over live system snapshots. The resulting graph can therefore serve as up-to-date, machine-queryable operational documentation of the local cloud.

Table 1 compares AFO with related semantic resources along the ontology-level dimensions most relevant to Arrowhead framework local-cloud modeling. Operationalization-specific aspects of our approach, such as SHACL validation, OWL reasoning over instantiated graphs, and timestamped snapshots, are evaluated separately in Section IV. The table shows that generic industrial ontologies provide reusable background semantics, and that prior Arrowhead framework-specific ontologies already capture important Arrowhead concepts, but the AFO is distinguished by its explicit focus on the runtime deployment structure and service dependencies of instantiated local clouds.

III. ARROWHEAD FRAMEWORK ONTOLOGY

An ontology provides a formally defined vocabulary and relations for a domain, enabling consistent interpretation and reasoning over data by both humans and machines. In

TABLE 1. Ontology-Level Comparison of AFO With Related Semantic Resources (See Section II for References)

Resource	AH-spec.	Runtime LC	Prov./ Cons.	Orch./ Gov.	Host/ Husk/ Server	Explicit constraints	Position relative to AFO
SSN/SOSA	N	N	N	N	N	N	generic sensing semantics
SAREF	N	N	P	N	N	N	device/service semantics
IOF/IDO	N	N	N	N	N	N	manufacturing ontology backbone
OPC UA → OWL work	N	N	N	N	N	P	asset/interface lifting
SAAS	N	N	N	N	N	P	semantic AAS layer
I4GO ArrowHead	Y	P	P	P	N	P	broader modular Arrowhead basis
Digital Reference Arrowhead	Y	P	P	P	P	P	closest prior Arrowhead resource
AFO	Y	Y	Y	Y	Y	Y	deployment-oriented runtime ontology

Y = Explicit and Central; P = Partial or Not Primary; N = Not a Main Focus. Workflow-specific aspects, such as SHACL validation, OWL reasoning over deployed instances, and timestamped snapshots are evaluated separately in Section IV.

this sense, it is *an explicit specification of a conceptualization* [37]. This section presents the AFO by outlining its main components and their relations.

To ensure globally unique and unambiguous identification, the semantic web uses internationalized resource identifiers (IRIs) to denote classes, properties, individuals, and ontologies. Since IRIs tend to be long, their common roots are shortened as prefixes, such as `owl:`¹ for OWL 2, `alc:`², and `afo:`³ for the AFO namespace. The AFO is implemented in OWL 2, a W3C-standard ontology language with a formal, model-theoretic semantics, and serialized in Turtle to provide compact and human-readable RDF data [10].

The AFO presented in this work is developed against the mbaigo realization of an arrowhead framework-inspired local cloud rather than against the full Eclipse arrowhead reference stack. Accordingly, the ontology reflects the systems currently instantiated in mbaigo. In the demonstrator considered here, the service registrar and orchestrator are operational core supporting systems, while the security layer is still evolving. The Certificate Authority (CA) represents the currently present security-related system in this implementation, but since authentication/authorization support is not yet fully developed in mbaigo, it is not treated here as a fully mandatory modeled subsystem.

A. ONTOLOGY SCOPE AND COMPETENCY QUESTIONS

The AFO is intended to provide a vocabulary and a set of structural constraints sufficient to represent the runtime structure of a local cloud, the systems it contains, and the service dependencies through which they interact.

The current scope of AFO is deployment topology, service dependencies, core-system roles, and temporal deployment traces. It does not yet model detailed control logic, continuous telemetry streams, QoS constraints, full authorization-policy semantics, or intercloud federation.

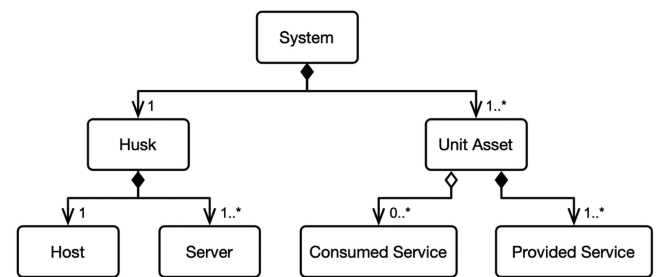


FIGURE 1. Architectural overview of a generic arrowhead framework system.

To guide the ontology design and to provide a basis for its subsequent evaluation, we define the following competency questions.

- 1) *CQ1*: Which systems provide a given service?
- 2) *CQ2*: Which consumed services are currently unsatisfied?
- 3) *CQ3*: On which host does the husk of a given system run?
- 4) *CQ4*: What is the operational status of a local cloud, i.e., resilient, degraded, or nonoperational?
- 5) *CQ5*: What is the temporal history of the device installed at a given functional location?

These competency questions operationalize the information needs that the AFO is intended to support. Section IV evaluates them through the instantiated knowledge graph using SHACL validation, OWL reasoning, and SPARQL queries. *CQ5* additionally relies on timestamped snapshots and the deployment-specific `alc: extension` ontology.

B. KEY FUNDAMENTAL CLASSES AND PROPERTIES

A local cloud is treated as a SoS composed of multiple systems that are distinct instances but follow a common structural pattern. Fig. 1 illustrates this generic arrowhead framework system structure. A system is composed of exactly one *Husk* and one or more *Unit Asset* instances. The husk is the system's software interface between the unit assets and the local cloud: it is composed of one *Host* and one or more *Servers* that listen for requests from consumers. Unit assets can be

¹ <http://www.w3.org/2002/07/owl#>

² <http://www.synecdoque.com/2025/alc#>

³ <http://www.synecdoque.com/2025/afo#>

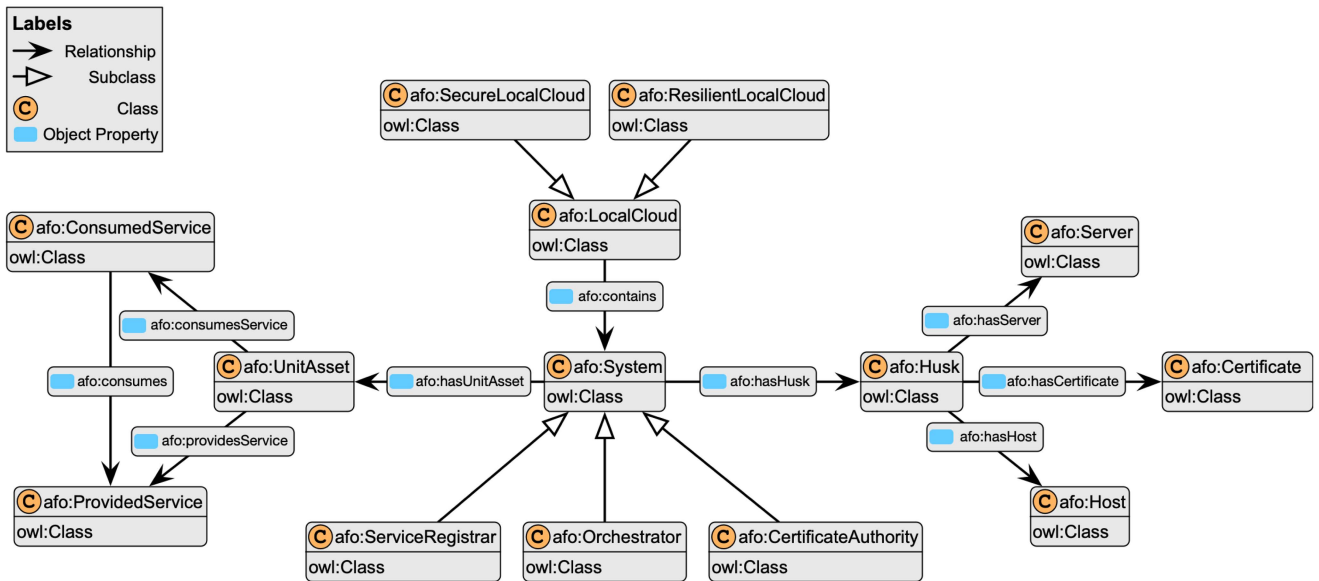


FIGURE 2. Graph view of the key fundamental classes and properties of the AFO.

hardware (e.g., sensors and actuators) or software (control software, databases). A unit asset carries out the system’s functions, which realize the services exposed via the husk. When needed, the system may also consume external services, again via the husk. The unit asset is composed of at least one *Provided Service* and aggregates any number of *consumed services*.

Building on this conceptual model, we define a core ontology that turns these parts into formal OWL 2 classes and object properties so they can be queried and reasoned over. The resulting core model is shown in Fig. 2 and serves as the semantic backbone used throughout the article.

As illustrated in Fig. 2, a local cloud is described as containing systems via the object property `afo:contains` (domain `afo:LocalCloud`, range `afo:System`). The inverse direction is made explicit by declaring `afo:isContainedIn` as the inverse of `afo:contains`. Similarly, `afo:System` is defined in terms of its aggregation: each system is related to exactly one `afo:Husk` and to one or more `afo:UnitAsset` instances via `afo:hasHusk` and `afo:hasUnitAsset`.

Following the arrowhead framework deployment model, the husk runs on a host and exposes protocol-specific servers [34]. In the ontology, `afo:Host` and `afo:Server` are modeled as classes and linked to `afo:Husk` via `afo:runsOnHost` and `afo:hasServer`. Finally, `afo:UnitAsset` captures the hardware or software component that provides and/or consumes services. These interactions are represented using `afo:providesService` and `afo:consumesService`, which link a unit asset to `afo:ProvidedService` and `afo:ConsumedService`, respectively, enabling queries in both directions through declared inverse properties.

C. AXIOMS AND INFERENCE

The arrowhead framework generic system design imposes structural constraints that are useful for validation and automated classification. The AFO makes this pattern machine-interpretable by formalizing it as OWL axioms, so reasoners can infer classifications and detect inconsistencies with the intended model. For example, a system may have many unit assets, whereas each unit asset belongs to exactly one system. Accordingly, `afo:hasUnitAsset` is paired with its inverse `afo:isUnitAssetOf`. We declare `afo:isUnitAssetOf` as *functional* (at most one system per unit asset) and constrain `afo:UnitAsset` to have at least one `afo:isUnitAssetOf` value. Together, these axioms encode the intended one-to-many pattern. If a unit asset is asserted to belong to multiple systems, a reasoner will either identify those systems as the same individual or, if they are declared distinct, report an inconsistency.

Service interactions are modeled explicitly. Unit assets relate to the exposed services through `afo:providesService` and `afo:consumesService`. We distinguish provided and consumed services by declaring `afo:ProvidedService` and `afo:ConsumedService` as disjoint subclasses of `afo:Service`. As Fig. 3 indicates, each unit asset is constrained to provide at least one service, while consuming services remains optional, reflecting that some systems can operate without external dependencies. Services also carry required metadata (e.g., endpoint URL, service definition, and subscribability) to support consistent discovery and orchestration.

On the interface side, the husk is required to refer to both a host and at least one server. In AFO, `afo:Husk` is constrained to have exactly one `afo:Host` via `afo:runsOnHost` and at least one `afo:Server` via

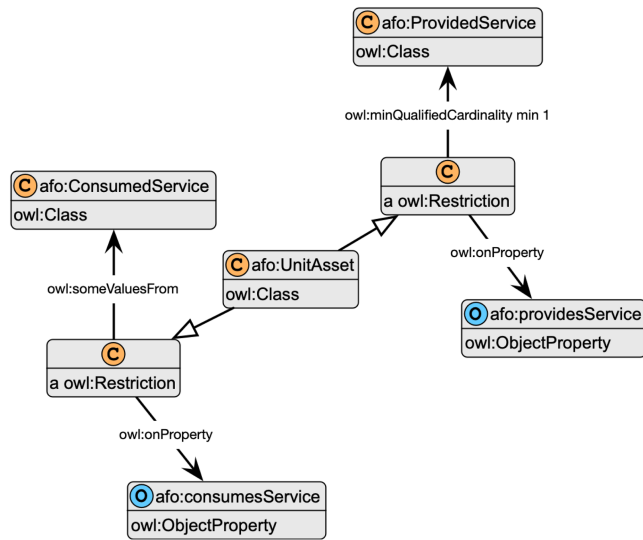


FIGURE 3. Qualified minimum cardinality axiom stating that each `afo:UnitAsset` *provides* at least one `afo:ProvidedService`.

`afo:hasServer`. Servers carry minimal protocol metadata (protocol identifier and port number) to support deployment and runtime configuration.

We call a local cloud *resilient* when it includes at least two instances of each mandatory core system, specifically the service registrar and the orchestrator (modeled as subclasses of `afo:System` in Fig. 2). In the case of the service registrar, one instance acts as leader while another serves as standby, enabling failover. If a local cloud additionally contains a CA system (a non-mandatory core system), it is considered *secure*. These notions are represented by the subclasses `afo:ResilientLocalCloud` and `afo:SecureLocalCloud`, which specialize `afo:LocalCloud` with additional constraints. The AFO models concrete system instances as individuals of `afo:System`. Instances are annotated with identifying deployment metadata (a name or role label) that distinguishes core-system types. By referencing this metadata in class axioms, the ontology allows automated classification of system instances into specific core-system classes.

Together, these classes, properties, and axioms make the AFO both structurally faithful to the Arrowhead framework and logically actionable, enabling automated classification, consistency checking, and downstream reasoning used throughout the rest of this work.

IV. REPLICABLE DEMONSTRATION

The ontology in Section III is only useful if it supports concrete engineering tasks in a running local cloud. This section therefore presents a replicable demonstration in which the AFO is instantiated from a deployed arrowhead framework SoS. We ingest the resulting RDF descriptions into GraphDB, validate them, and apply OWL reasoning to materialize entailments. We then answer diagnostic and traceability questions through qualitative query interactions. The examples illustrate how stakeholders can retrieve, combine, and

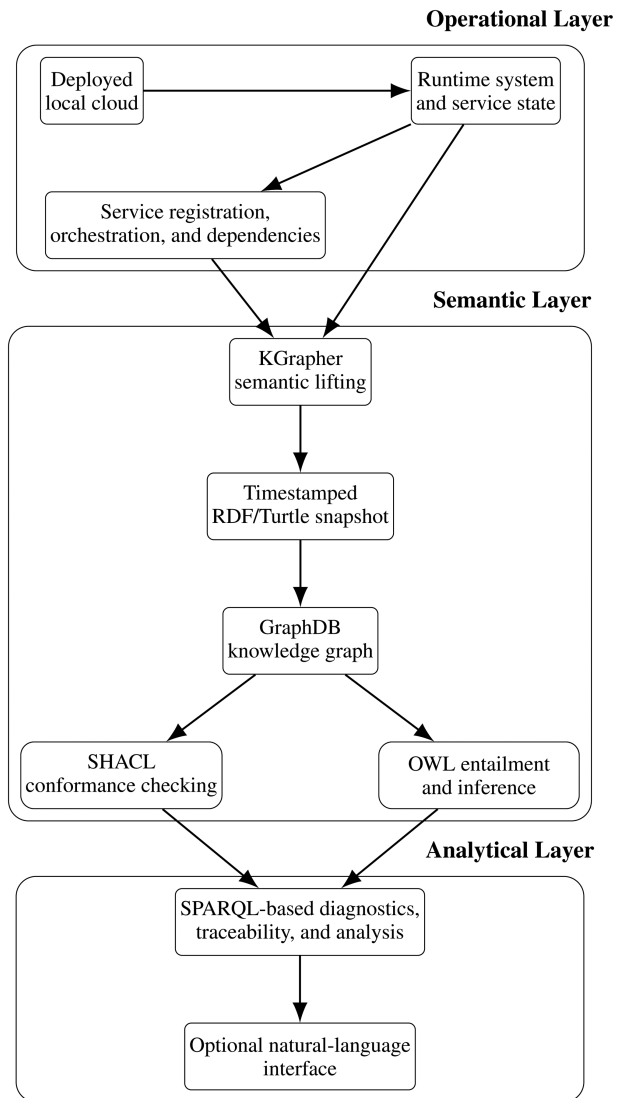


FIGURE 4. End-to-end workflow of the proposed approach. A deployed `mbaigo` local cloud is semantically lifted by the KGrapher into timestamped RDF/Turtle snapshots, which are ingested into GraphDB and processed through SHACL conformance checking and OWL reasoning to support diagnostic querying, traceability, and optional natural-language access.

interpret facts that are difficult to obtain from isolated services or conventional documentation. Fig. 4 summarizes this end-to-end workflow, from the deployed local cloud and KGrapher-based semantic lifting to RDF ingestion, SHACL validation, OWL reasoning, and query-based analysis.

A. SETUP

In the workflow shown in Fig. 4, the climate-control demonstrator constitutes the deployed local cloud from which runtime descriptions are extracted and transformed into a queryable knowledge graph. The demonstrator is a distributed climate-control system composed of sensor, actuator, and thermostat systems. It can be deployed on one or more Raspberry Pi hosts on the same internet protocol (IP) network. The resulting SoS consists of at least six systems: the Service Registrar, the Orchestrator, the

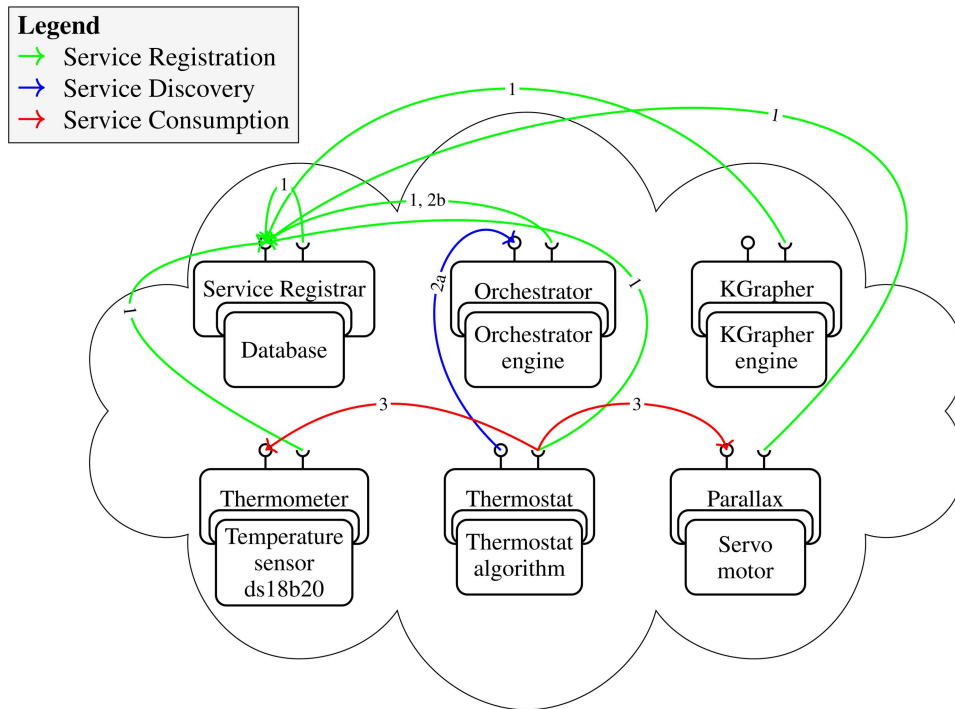


FIGURE 5. Climate-control local cloud.

DS18B20, the Parallax, the Thermostat, and the KGrapher. These systems provide the runtime state that the KGrapher semantically lifts into timestamped RDF/Turtle snapshots. The DS18B20 system’s unit asset is a temperature sensor, specifically a 1-wire DS18B20 probe. The Parallax system emulates a valve using a Parallax servo motor driven by a pulsewidth-modulated control signal. The Thermostat system’s unit asset is a PID controller that consumes the temperature and valve-position services. The demonstrator can run without the actuator because its physical presence is not enforced by the service interface. It can partially run without the sensor, but it will report the missing dependency.

The AFO repository provides the ontology and SHACL shapes used in this work.⁴ Replicating the deployed demonstrator additionally requires the mbaigo base library,⁵ and the mbaigo systems release used in this study, *v0.1.0-alpha.3*.⁶

A minimal knowledge-graph demonstrator comprises *esr*, *orchestrator*, and *kgrapher*; the climate-control demonstrator further uses *ds18b20*, *parallax*, and *thermostat*.

Optional systems include a CA and a collector. The CA supports a public-key infrastructure in which each system generates a private key at startup and requests a signed certificate from the CA. The Collector logs signals using InfluxDB as its unit asset, i.e., a time-series database.

The local cloud setup is illustrated in Fig. 5. The source code can also be compiled for other platforms, although the

sensor and actuator require the Raspberry Pi’s hardware interfaces. When using multiple hosts, resiliency aspects can be evaluated by stopping selected systems or failing entire hosts.

The KGrapher system offers the *cloudgraph* service, which assembles an RDF knowledge graph from the data exposed by systems in the local cloud and returns it as Turtle payload. The KGrapher also exports the resulting payload to its unit asset, a graph database, where it is ingested. In our setup, the graph database is Ontotext GraphDB, which can run on any host and is available in a free-to-use edition.⁷ For each invocation of the *cloudgraph* service, the KGrapher produces a local-cloud description that is ingested into GraphDB as a timestamped named graph. In the demonstrator, the KGrapher generates a complete SoS snapshot in approximately 10 ms. This time covers two stages of inter-system communication. First, the KGrapher issues a GET request to the service registrar to obtain the current set of active systems. Second, the KGrapher issues one GET request to each active system to retrieve its RDF description, and these requests are executed concurrently to reduce end-to-end latency as the number of systems grows. The reported snapshot time excludes ingestion into GraphDB. For the small SoS considered here, ingestion and parsing were not perceived to be a performance bottleneck. During ingest, GraphDB performs syntactic validation using its RDF parser and aborts the transaction if parsing fails. This includes errors, such as malformed prefixes or incorrectly declared instances, in which case GraphDB returns a validation report. OWL inference is applied only after the data parses successfully. Syntactic validation at ingest provides immediate feedback,

⁴ <https://github.com/oskwin/Arrowhead-Framework-Ontology>

⁵ <https://github.com/sdoque/mbaigo>

⁶ <https://github.com/sdoque/systems/releases/tag/v0.1.0-alpha.3>

⁷ <https://www.ontotext.com/products/graphdb/>

```

9f850b7c0e8a4ecb96f599d993c75c2b2722379 a sh:ValidationResult;
sh:focusNode <http://www.synecdoque.com/1cloud/AlphaCloud_wintsson_telegrapher>;
rsx:shapesGraph rdf4j:SHACLShapeGraph;
sh:resultPath <http://www.synecdoque.com/2025/afo#hasHusk>;
sh:sourceConstraintComponent sh:QualifiedMinCountConstraintComponent;
sh:resultSeverity sh:Violation;
sh:sourceShape ...9fce2ad6-8dd9-499c-a8b0-57f9bf1ff4c5-5 .

...9fce2ad6-8dd9-499c-a8b0-57f9bf1ff4c5-5 a sh:PropertyShape;
sh:path <http://www.synecdoque.com/2025/afo#hasHusk>;
sh:qualifiedValueShape ...59c081ad-a7d5-4dc6-916f-cb65f9c696ad-6;
sh:qualifiedMinCount 1 .

...59c081ad-a7d5-4dc6-916f-cb65f9c696ad-6 a sh:NodeShape;
sh:class <http://www.synecdoque.com/2025/afo#Husk> .

...9f850b7c0e8a4ecb96f599d993c75c2b2722380 a sh:ValidationResult;
sh:focusNode <http://www.synecdoque.com/2lcloud/AlphaCloud_wintsson_telegrapher_Bathroom>;
rsx:shapesGraph rdf4j:SHACLShapeGraph;
sh:resultPath <http://www.synecdoque.com/2025/afo#providesService>;
sh:sourceConstraintComponent sh:QualifiedMinCountConstraintComponent;
sh:resultSeverity sh:Violation;
sh:sourceShape ...b69ac05e-ec98-4385-b577-105b41e7d741-9 .

...b69ac05e-ec98-4385-b577-105b41e7d741-9 a sh:PropertyShape;
sh:path <http://www.synecdoque.com/2025/afo#providesService>;
sh:qualifiedValueShape ...b48f5fc4-4442-4909-a212-3362c86f42b4-10;
sh:qualifiedMinCount 1 .

...b48f5fc4-4442-4909-a212-3362c86f42b4-10 a sh:NodeShape;
sh:class <http://www.synecdoque.com/2025/afo#Service> .
    
```

FIGURE 6. Result of a SHACL validation performed in GraphDB. In the figure, each validation result is defined as a text block followed by a property block and a node block.

prevents downstream reasoning failures, and protects repository integrity.

B. INFERENCE AND VALIDATION

When ingestion is performed with the AFO loaded in GraphDB, the resulting knowledge graph becomes semantically richer and supports more expressive queries and reasoning. The ontology contributes structure through class hierarchies, inverse properties, and additional axioms that increase connectivity by turning many unidirectional links into logically bidirectional relations.

Following the syntax validation during ingestion (see Section IV-A), the data is subjected to model-level validation using SHACL shapes. Rather than validating OWL axioms directly, we verify that the ingested instance data conforms to a set of SHACL constraints that operationalize data-quality requirements aligned with the ontology. OWL axioms provide schema-level semantics under the open-world assumption, whereas SHACL provides a controlled closed-world check that can reveal missing information and prevent silent drift in the instance model.

The shapes target nodes by class, property paths, or SPARQL-based scopes to specify where each constraint applies. They express requirements such as mandatory object properties, cardinality constraints, allowed datatypes, and permitted node kinds. Validation produces a SHACL report that lists, for each violation, the focus node, the offending path, the violated shape, and a severity level together with a human-readable message, as shown in Fig. 6. In this example, the report identifies the Telegrapher system as the source of the violation because it lacks a husk and provides no service. In our workflow, SHACL validation is treated as a policy gate rather than a numerical performance metric: the absolute number of violations depends on how many faults are

```

1 PREFIX afo: <http://www.synecdoque.com/2025/afo#>
2 PREFIX onto: <http://www.ontotest.com/>
3
4 CONSTRUCT { ?s ?p ?o }
5 FROM NAMED onto:implicit
6 FROM NAMED onto:explicit
7 WHERE {
8   GRAPH onto:implicit { ?s ?p ?o }
9   GRAPH onto:explicit { ?s ?p ?o }
10  FILTER( STRSTARTS(STR(?p), STR(afo:)) )
11 }
12
13 FILTER( STRSTARTS(STR(?s), STR(afo:)) )
    
```

FIGURE 7. Identical query on two different states of the knowledge graph: explicit triples versus explicit+entailed triples from the AFO.

present or injected, and is therefore not meaningful in isolation. Instead, we assess the validation layer by whether it 1) detects representative classes of deployment faults and 2) localizes them to actionable focus nodes and paths in the report. Accordingly, we model minimal counterexamples for several recurring fault classes, such as missing mandatory deployment links (a system without a husk or without any provided service), and metadata policy violations (datatypes for service endpoints or missing data). Violations are treated as blocking errors for downstream processing, whereas warnings are retained as nonblocking diagnostics that provide immediate, actionable feedback. Shapes are maintained in a separate shapes graph under version control so that changes to the constraint set remain auditable and reproducible. Only once the data conforms to these shapes do we proceed to OWL reasoning, ensuring that inference operates over a consistent and policy-compliant graph and avoiding downstream dependency traces. The number of reported violations is not a result in itself, but exemplifies how validation operationalizes deployment policy by producing actionable, localized diagnostics that gate the subsequent OWL reasoning step.

After SHACL-conformant snapshots are ingested, we apply OWL 2 rule language (OWL 2 RL) reasoning to infer new knowledge about the validated information. The reasoner materializes entailed statements to increase completeness, improve semantic consistency, and support analytics, compliance checks, and decision-making. Fig. 7 contrasts the explicit graph with the graph obtained after materialization of entailments. At one point, the export contains 178 asserted statements, while reasoning materializes an additional 54 inferred statements (31.4% increase), primarily due to 1) type propagation along class hierarchies; 2) inverse-property completion; and 3) rule-based entailments over service relations. The reasoning infers links that are not made explicit by the exporter, such as `?system rdf:type afo:ServiceRegistrar` based on definitional axioms over metadata. For example, given the asserted pattern

```

?system rdf:type afo:System;
?system afo:hasName "serviceregistrar"
    
```

The reasoner entails the classification

```

?system rdf:type afo:ServiceRegistrar
    
```



FIGURE 8. Result of a reasoning process in Protégé; the row marked in yellow represents inferred knowledge.

and creates more precise querying capabilities through classifications. Reasoning also specifies certain inverse properties, when available, which normalizes deployment queries regardless of the direction used by the sources. In our setup, the reasoning is performed on ingestion which GraphDB completed in less than one second on the Raspberry Pi 5, and the resulting snapshot graphs remain fully queryable with latency coming from standard SPARQL evaluation rather than inference.

The 54 inferred statements are not reported merely as a raw increase in graph size. Their practical relevance is that three of the five competency questions (CQ1, CQ2, and CQ4) depend on or benefit from the validated and entailed graph, whereas only one (CQ3) can be answered from asserted deployment relations alone and one (CQ5) requires timestamped named graphs. The added entailments therefore increase effective query coverage and reduce the need for deployment-specific query logic.

Finally, to verify schema consistency and inspect the ontology's interaction with the data, the knowledge graph is exported from GraphDB and imported into Protégé.⁸ There, we treat Hermit reasoning as a schema-level consistency verification. Domain adequacy is evaluated separately through competency questions and use-case-based verification. Fig. 8 shows the result of this reasoning, where the row marked in yellow represents inferred knowledge.

C. DIAGNOSTICS

We argue that robust interpretation and diagnosis require explicit contextual knowledge to support automated reasoning. In the climate-control SOS, no individual constituent system is designed to diagnose global faults, because each system only exposes its own services and local state. However, once run-time bindings and dependencies are captured in the knowledge graph and enriched with AFO semantics, SoS-level diagnosis becomes possible through queries. If a sensor or actuator system is shut down and therefore stops providing a required service, the knowledge graph reflects that the corresponding consumed service is no longer satisfied. This allows a diagnostic query to identify missing providers and violated dependencies at the level of the composed SoS. Listing 10 in the Appendix lists a SPARQL query that returns all unsatisfied service dependencies in the current deployment. For each violation, the query reports the consuming system, the missing

consumed service, and the service definition that could not be resolved to a currently available provider. In our demonstrator, stopping the valve system causes the thermostat's dependency on the valve-position service to become unsatisfied, and the query flags this violation immediately after the next snapshot is ingested. This makes the failure mode observable and explainable at the SoS level, even though the constituent systems were not originally engineered for diagnosis as a stand-alone function.

D. DIGITAL THREAD

Diagnostics are only one application of knowledge graphs, and digital threads are another. A digital thread is an integrated and authoritative flow of data that links a product's or system's artifacts, models, and decisions across its lifecycle. It spans concept and design, manufacturing, operation, and decommissioning. This continuity enables traceability and reuse by allowing artifacts and decisions to be queried as one coherent whole.

For example, services such as milling, drilling, or heating incur both financial and environmental costs. It is often useful to determine the cost and impact associated with producing a specific product, which can be obtained by querying the knowledge graph. In industrial practice, this supports environmental assessment, such as LCA, and economic evaluation, such as LCC [38]. In the climate-control demonstrator, these costs are associated with the Thermostat system's set-point service. Another illustration of a digital thread is product genealogy and traceability. A DPP can be viewed as a machine-readable certificate of origin. Consider a mixing service represented in the knowledge graph together with relationships to the raw-material batches used during that process [39]. With such Semantic Web representations, new DPP requirements can be addressed through queries over a shared graph rather than bespoke interfaces.

In the climate-control use case, a basic digital thread is the functional location of an installed device. A functional location is a hierarchically arranged and uniquely identifiable place in a plant where technical functions are performed and equipment can be installed, as defined in the IEC 81346 series on structuring principles and reference designations [40]. For the demonstrator, this thread can be phrased as the question: "What is the history of the temperature sensor installed in the kitchen?"

The KGrapher pushes each local-cloud snapshot to GraphDB as a timestamped named graph, enabling changes to be traced over time. Starting from a working climate-control setup, we invoke the `cloudgraph` service, then stop the DS18B20 system and capture a second snapshot. We then restore operation by restarting the system, either with the same temperature sensor or with a replacement sensor, and capture a third snapshot. Using the SPARQL query in Listing 12 in the Appendix, we reconstruct the sensor history at the functional location, as illustrated in Fig. 9. The figure shows three constructed timestamp nodes that connect to the sensor individual present at each snapshot. The highlighted objects correspond to two distinct physical sensors with different identifiers. In

⁸ <https://protege.stanford.edu/>

	subject	predicate	object
1	urn:2026-02-20T14:07:38	rdfs:type	afo:FunctionalLocation
2	urn:2026-02-20T14:07:38	alc:hasLocation	alc:Kitchen
3	urn:2026-02-20T14:07:38	afo:hasUnitAsset	alc:AlphaCloud_jeppcorn_ds18b20_28-0000f02986f
4	alc:AlphaCloud_jeppcorn_ds18b20_28-0000f02986f	rdfs:label	"28-0000f02986f"
5	urn:2026-02-23T08:41:22	rdfs:type	afo:FunctionalLocation
6	urn:2026-02-23T08:41:22	alc:hasLocation	alc:Kitchen
7	urn:2026-02-23T08:44:37	rdfs:type	afo:FunctionalLocation
8	urn:2026-02-23T08:44:37	alc:hasLocation	alc:Kitchen
9	urn:2026-02-23T08:44:37	afo:hasUnitAsset	alc:AlphaCloud_jeppcorn_ds18b20_28-0000f033350
10	alc:AlphaCloud_jeppcorn_ds18b20_28-0000f033350	rdfs:label	"28-0000f033350"

FIGURE 9. Resulting triples of a SPARQL query constructing a digital thread of a location.

TABLE 2. Competency Questions, AFO Constructs Involved, Evaluation Mechanisms, and Supporting Evidence in the Demonstrator

CQ	Key AFO constructs	Mechanism used	Evidence in paper
CQ1	afo:System, afo:UnitAsset, afo:ProvidedService, afo:hasUnitAsset, afo:providesService	Asserted graph + OWL reasoning + SPARQL	Provider relations are represented in the instantiated graph and become more directly queryable after classification and inverse-property completion; see Section IV-B, Listing 4 in the Appendix and Figure 7.
CQ2	afo:ConsumedService, afo:consumes, afo:providesService, afo:hasUnitAsset	SPARQL over validated and entailed graph	Demonstrated in Section IV-C through the diagnostic query for unsatisfied dependencies; see Listing 1 in the Appendix.
CQ3	afo:System, afo:Husk, afo:Host, afo:hasHusk, afo:runsOnHost	Asserted graph + SPARQL	Deployment context is explicitly represented in the graph through the system–husk–host relations introduced in Section III and instantiated in the local-cloud snapshot. See Listing 5 in the Appendix.
CQ4	afo:LocalCloud, afo:ServiceRegistrar, afo:Orchestrator, afo:ResilientLocalCloud	OWL classification + SPARQL aggregation	Operational status is evaluated through counts of redundant core systems and their host placement; see Listing 2 in the Appendix.
CQ5	afo:FunctionalLocation, afo:atTime, afo:hasUnitAsset, alc:hasLocation	Timestamped named graphs + SPARQL CONSTRUCT	Demonstrated in Section IV-D through reconstruction of the sensor history at a functional location; see Listing 3 in the Appendix and Figure 9.

the snapshot where the sensor was offline, the corresponding timestamp node has no afo:hasUnitAsset link, which is consistent with the observed fault. In the final snapshot, the timestamp node links to the replacement sensor, indicating that the system returned to an operational state.

Taken together, the preceding subsections address the competency questions introduced in Section III. Table 2 summarizes which AFO constructs are involved in each question, which mechanism is used to answer it, and where the supporting evidence appears in the article. This summary also provides the basis for the optional natural-language access discussed in the next section.

E. PRELIMINARY NATURAL-LANGUAGE ACCESS

Interacting with RDF data through formal query languages, such as SPARQL, can present a steep learning curve for many users. We therefore explored GraphDB’s *Talk to Your Graph* interface as an optional natural-language front end to the instantiated AFO knowledge graph. In our prototype, simple

competency-question-style prompts could be translated into useful SPARQL queries, but the local LLM setup was not reliable enough for deeper dependency-traversal questions. We therefore report this component as a preliminary usability exploration rather than as a validated operator-facing result.

V. DISCUSSION

Businesses pursue digital transformation to gain a deeper understanding of their operations, to support better decision-making, and to comply with legal requirements, such as DPP and environmental reporting. In heterogeneous industrial landscapes, these goals are difficult to achieve because data and functionality are distributed across many independently developed systems. The results in this article indicate that semantic web technologies can provide a practical backbone for integrating such landscapes without replacing the underlying infrastructure. By exporting existing deployment and engineering artifacts into a knowledge graph and enriching

them with formal semantics, operators obtain a unified and queryable representation of the running SoS.

A key observation concerns how a SoS is composed in practice. Constituent systems are typically designed and deployed as standalone units, and they are initially unaware of the specific other systems with which they will interact. In service-oriented architectures, composition happens at run time through discovery and orchestration, and bindings can change as systems appear, disappear, or are replaced. This late binding is precisely what enables flexible composition, but it also makes global behavior harder to predict and analyze from local system logs or static documentation. In our climate-control demonstrator, the desired behavior emerges only when multiple systems are bound together into a closed control loop. No single component embodies climate control in isolation, yet the composed SoS exhibits it as an emergent operational capability. The knowledge graph constructed by the KGrapher system captures this run-time structure, and the AFO makes the structure explicit enough to analyze.

The same mechanism also enables an emergent diagnostic capability. None of the constituent systems were designed to answer SoS-level questions, such as which dependencies are violated or which systems are affected by a failing service. Once bindings, roles, and constraints are represented explicitly, the SoS becomes diagnosable through queries over the integrated graph. OWL reasoning strengthens this effect by materializing implied relationships, propagating types, and completing inverse links, which improves query coverage and stability. SHACL complements this by providing a configurable validation layer that can flag incomplete or inconsistent configurations before they propagate into operational decisions. Together, explicit semantics, validation, and inference turn an implicit run-time integration into an explainable operational model.

The research questions posed in the introduction can now be addressed explicitly. (Q1) is supported by OWL-based inference, which derives additional knowledge about systems, services, and their relationships beyond what is asserted in the exported configuration. (Q2) is supported by SHACL-based validation, which checks instance-level data quality and integrity constraints aligned with the ontology, and produces actionable reports when constraints are violated. (Q3) is supported by the demonstrated diagnostic and impact-analysis queries, where inferred dependency knowledge exposes missing providers, violated requirements, and degraded modes that are not apparent from any single system. (Q4) is addressed by the fact that the approach is compatible with edge participation without requiring full semantic processing on constrained devices. Edge systems contribute structured snapshots through existing service interfaces, while storage, validation, and reasoning can be placed where resources and governance are available, such as on an on-premise server or a plant-level node. (Q5) is addressed in this article through the digital-thread demonstration, where timestamped snapshots and contextual links enable reconstruction of the history of

a device at a functional location. This establishes traceability over time in the deployed local cloud. Broader lifecycle applications, such as provenance, LCA, and LCC, are conceptually compatible with the same mechanism but are not directly evaluated in the present demonstrator.

From a deployment perspective, a central advantage of the approach is that it is largely nonintrusive. The AFO adds a semantic layer on top of existing systems, protocols, and engineering tooling rather than replacing them. This enables incremental adoption, where a project can begin with read-only export and query, and later introduce validation, reasoning, and operator-facing interfaces as the benefits become clear. In this sense, semantic enrichment functions as an overlay that increases observability and coherence without demanding a wholesale technology migration. In the present demonstrator, ingesting exported snapshots into GraphDB was fast (2000+ statements in less than one second), and data loading did not become a practical bottleneck. These results demonstrate feasibility for the demonstrator size studied here, but they do not by themselves establish broader scalability. A more systematic evaluation over larger local-cloud configurations and alternative deployment sizes remains future work.

To capture deployment-specific context, KGrapher uses the `alc:` prefix, which denotes the arrowhead local cloud (ALC) ontology. ALC is intended as a stakeholder-defined extension ontology that classifies assets and concepts specific to a given deployment. In the climate-control example, classes, such as `alc:Room` can represent the locations, in which sensors and actuators are installed. In an industrial plant, analogous classes can represent equipment identifiers, tags, or functional locations used in engineering and maintenance practice.

Several limitations and tradeoffs remain. First, ontologies and knowledge graphs are not self-maintaining assets, and long-term value depends on stewardship and integration into engineering workflows. The practical issue is not ownership in a proprietary sense, but sustained governance of scope, versioning, and alignment with evolving framework releases and local extensions. In an open-source setting, this can be handled through shared maintenance practices, transparent change control, and sustained funding mechanisms that support collective benefit. Second, SHACL interacts in subtle ways with the open-world assumption of RDF and OWL. A missing violation does not guarantee completeness, and strict enforcement can reject snapshots that are operationally meaningful because they capture misconfigurations, temporary workarounds, or undocumented changes that later explain failures. A pragmatic strategy is therefore to apply SHACL selectively as a diagnostic and policy layer, while still allowing historically imperfect snapshots to be stored for root-cause analysis and long-term trend studies. A further practical limitation concerns the natural-language interface. In our prototype, the local LLM setup (Ollama with Llama 3.1) was not sufficiently precise to be treated as fully reliable for

operator-facing use. It could answer shallow inventory-style questions, but it became inconsistent for deeper graph-traversal queries and was sensitive to prompt and configuration details. We therefore treat natural-language access as a preliminary usability extension rather than as a validated contribution of the present work. A more systematic evaluation of question coverage, translation correctness, and answer reliability is needed before such an interface can be considered production-ready.

Security and confidentiality are also central concerns for production adoption. If machines can derive meaningful insight from operational knowledge, an attacker or competitor could potentially derive the same insight if the knowledge graph or its services are exposed. This risk calls for strong authentication, authorization, and network segmentation around the repository, its endpoints, and any natural-language interface built on top of it. In addition, practical deployments may benefit from separating internal operational graphs from filtered, purpose-specific views that are shared with external stakeholders for compliance or supply-chain collaboration.

Finally, semantic interoperability beyond a single enterprise remains a broader challenge. Within one organization, a local ontology can often provide sufficient shared meaning for diagnosis and traceability. Across organizations, however, shared meaning requires alignment to common industrial upper ontologies and well-defined mappings. The AFO is designed to reference and reuse other ontologies, and lightweight mapping layers can relate arrowhead concepts to broader industrial vocabularies when cross-organizational threads are required.

The demonstrator presented in Section IV should be interpreted as a proof-of-concept evaluation on an author-developed arrowhead framework local cloud. Its main purpose is to assess if the AFO captures the intended deployment semantics under different conditions, such as functional operation, disturbance, and maintenance. It does not constitute external validation across independently developed Arrowhead framework deployments or third-party evaluators; such broader validation remains as future work.

The present evaluation is not based on a single static snapshot. Rather, it exercises the ontology and pipeline across several representative operating conditions: a healthy deployment, malformed instance data detected through SHACL, inference-based enrichment of core-system roles, degraded operation caused by missing service providers, and recovery/maintenance scenarios captured through timestamped digital-thread reconstruction. In this sense, the demonstrator functions as a structured scenario-based proof of concept rather than as a one-shot example.

Overall, the results suggest that the benefits of introducing the AFO outweigh the identified drawbacks. The semantic backbone makes run-time SoS composition explicit, enables inference and validation over system and service relationships, and supports operator-relevant tasks, such as diagnosis, impact analysis, and digital-thread

reconstruction. With appropriate governance, selective validation strategies, and strong security controls, the approach provides a practical path toward more transparent, analyzable, and evolvable arrowhead-based automation solutions.

VI. CONCLUSION

This article introduced the AFO as a semantic model that makes the structure and run-time relationships of ALC explicit and machine-interpretable. The AFO formalizes core concepts, such as local clouds, systems, husks, unit assets, and services, and it encodes architectural constraints as OWL axioms that support automated classification and consistency checking. By instantiating the ontology from a running deployment using the KGrapher system and materializing the resulting knowledge graph in GraphDB, we demonstrated how Semantic Web technologies can be applied as a non-intrusive layer on top of existing industrial infrastructure.

The demonstration showed that explicit semantics enable three practical outcomes. First, OWL reasoning enriches deployment snapshots with inferred knowledge, increasing query coverage and making multihop dependencies directly accessible. Second, SHACL validation provides an actionable data-quality layer that detects misconfigurations and drift and prevents unreliable snapshots from propagating into operational decisions. Third, the same semantic backbone supports SoS-level diagnosis and digital-thread reconstruction, including traceability over time through timestamped named graphs and functional-location history. We additionally explored how competency questions grounded in the AFO might be exposed through natural-language interaction, although the present prototype was not yet reliable enough for real operational use.

These results address the research questions posed in the introduction by demonstrating that a SoS can infer new knowledge about its structure, validate incoming information for integrity, and become diagnosable through queries over an integrated semantic representation. The approach is feasible for edge-to-cloud deployments because constrained systems only need to expose structured interfaces, while semantic storage, validation, and reasoning can be placed where compute and governance are available. Moreover, the same knowledge-graph representation supports digital threads for operational traceability, demonstrated here through functional-location history. Broader lifecycle applications, such as product provenance, LCA, and LCC, are promising extensions of the approach rather than directly evaluated outcomes of the present demonstrator.

Future work will focus on expanding the ontology to cover additional arrowhead services and operational concerns, and on strengthening interoperability by aligning AFO concepts with industrial reference ontologies and domain standards. We also plan to refine validation strategies that balance strict constraint checking with the preservation of historically meaningful “imperfect” snapshots for root-cause analysis. Finally, because operational knowledge graphs can reveal

sensitive business insights, further work will investigate security controls and controlled disclosure mechanisms for sharing purpose-specific views with external stakeholders.

APPENDIX A SPARQL LISTINGS

PREFIX (used below):

```
PREFIX afo: <http://www.synecdoque.com/2025/afo#>
PREFIX alc: <http://www.synecdoque.com/2025/alc#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
```

```
SELECT DISTINCT
  ?consumerSystem ?consumerName
  ?consumerUnit
  ?consumedService
  ?providerService
  ?providerSystem ?providerName
WHERE {
  # Healthy snapshot: normal temperature trend
  GRAPH <urn:snapshots:2025-12-17T08:31:49Z> {
    # A system with a unit that consumes some
    # service
    ?consumerSystem a afo:System ;
      afo:hasName ?consumerName ;
      afo:hasUnitAsset ?consumerUnit .
    ?consumerUnit afo:consumesService ?
      consumedService .
    ?consumedService a afo:ConsumedService ;
      afo:consumes ?providerService .
    # That provider service was actually provided
    # by some system
    ?providerUnit afo:providesService ?
      providerService .
    ?providerSystem afo:hasUnitAsset ?providerUnit
      ;
      afo:hasName ?providerName .
  }
  # Faulty snapshot: irregular temperature trend
  GRAPH <urn:state:current> {
    ?consumerSystem a afo:System ;
      afo:hasUnitAsset ?consumerUnit .
    ?consumerUnit afo:consumesService ?
      consumedService .
  }
  FILTER NOT EXISTS {
    GRAPH <urn:state:current> {
      ?providerUnit2 afo:providesService ?
        providerService .
      ?providerSystem2 afo:hasUnitAsset ?
        providerUnit2 .
    }
  }
}
```

LISTING 1: Diagnostic query for missing service dependencies

```
SELECT
  ?cloud
  ?srCount ?srHostCount
  ?orCount ?orHostCount
  ?status
WHERE {
  # --- Service Registrar counts per local cloud
  ---
  {
    SELECT ?cloud
      (COUNT(DISTINCT ?sr) AS ?srCount)
      (COUNT(DISTINCT ?srHost) AS ?srHostCount)
    WHERE {
      ?sr rdf:type afo:ServiceRegistrar ;
        afo:isContainedIn ?cloud ;
        afo:hasHusk ?srHusk .
      ?srHusk afo:runsOnHost ?srHost .
    }
    GROUP BY ?cloud
  }
  # --- Orchestrator counts per local cloud ---
  {
    SELECT ?cloud
      (COUNT(DISTINCT ?or) AS ?orCount)
      (COUNT(DISTINCT ?orHost) AS ?orHostCount)
    WHERE {
      ?or rdf:type afo:Orchestrator ;
        afo:isContainedIn ?cloud ;
        afo:hasHusk ?orHusk .
      ?orHusk afo:runsOnHost ?orHost .
    }
    GROUP BY ?cloud
  }
  # --- Classify overall resilience state ---
  BIND(
    IF(
      ?srCount >= 2 && ?srHostCount >= 2 &&
      ?orCount >= 2 && ?orHostCount >= 2,
      "resilient",
      IF(
        ?srCount >= 1 && ?orCount >= 1,
        "degraded",
        "non-operational"
      )
    ) AS ?status
  )
  # Optional: restrict to a specific cloud, e.g.
  # AlphaCloud
  # FILTER(?cloud = alc:AlphaCloud)
}
```

LISTING 2: SPARQL query that evaluates the resilience status of a local cloud (resilient vs. degraded) based on redundant Service Registrar and Orchestrator deployments.

```

CONSTRUCT {
  ?dt a afo:FunctionalLocation ;
  afo:atTime ?t ;
  alc:hasLocation ?floc ;
  afo:hasUnitAsset ?ua .

  ?ua rdfs:label ?uaShort .
}
WHERE {
  VALUES (?floc ?system) {
    (alc:Kitchen alc:AlphaCloud_jeppcorn_ds18b20)
  }
  {
    SELECT DISTINCT ?g
    (STRAFTER(STR(?g), "urn:snapshots") AS ?tp)
    (STRBEFORE(?tp, "Z") AS ?ts)
    (xsd:dateTime(?ts) AS ?t) # same as above but
    in xsd:dateTime
  }
  WHERE {
    GRAPH ?g { ?s ?p ?o }
    FILTER (STRSTARTS(STR(?g), "urn:snapshots:"))
  }
}
BIND( IRI(CONCAT("urn:", ?ts)) AS ?dt )
OPTIONAL {
  GRAPH ?g {
    ?system afo:hasUnitAsset ?ua .
    ?ua alc:hasLocation ?floc .
  }
  # local name (everything after the last / or #)
  BIND(REPLACE(STR(?ua), "^.*[/#]", "") AS ?
    uaLocal)

  # strip the ds18b20 prefix
  BIND(REPLACE(?uaLocal, "^
    AlphaCloud_jeppcorn_ds18b20_", "") AS ?
    uaShort)
}
}

```

LISTING 3: SPARQL CONSTRUCT query used to reconstruct the temporal history of the sensor serving the thermostat's functional location from timestamped cloud snapshots.

```

SELECT DISTINCT ?providerSystem ?providerUnit ?
  service
WHERE {
  ?providerSystem a afo:System ;
    afo:hasUnitAsset ?providerUnit .
  ?providerUnit afo:providesService ?service .
  ?service a afo:ProvidedService ;
    afo:hasServiceDefinition "temperature" .
}

```

LISTING 4: SPARQL query used to find the provider/providers of a given service.

```

SELECT DISTINCT ?system ?husk ?host
WHERE {
  VALUES ?system { alc:AlphaCloud_jeppcorn_ds18b20
  }
  ?system afo:hasHusk ?husk .
  ?husk afo:runsOnHost ?host .
}

```

LISTING 5: SPARQL query used to find the host of a given system.

REFERENCES

- [1] International Data Corporation (IDC), "Worldwide IDC global data-sphere forecast, 2025–2029." International Data Corporation (IDC), 2025, Accessed: Jan. 20, 2026; IDC study abstract page. [Online]. Available: <https://my.idc.com/getdoc.jsp?containerId=US53363625>
- [2] T. Berners-Lee, J. Hendler, and O. Lassila, "The semantic web," *Sci. Amer.*, vol. 284, no. 5, pp. 34–43, May 2001. [Online]. Available: <https://lassila.org/publications/2001/SciAm.html>
- [3] S. Legg and M. Hutter, "Universal intelligence: A definition of machine intelligence," *Minds Machines*, vol. 17, no. 4, pp. 391–444, 2007, doi: [10.1007/s11023-007-9079-x](https://doi.org/10.1007/s11023-007-9079-x). [Online]. Available: <https://link.springer.com/article/10.1007/s11023-007-9079-x>
- [4] P. Lewis et al., "Retrieval-augmented generation for knowledge-intensive NLP tasks," in *Proc. 34th Int. Conf. Neural Inf. Process. Syst.*, 2020, pp. 9459–9474.
- [5] International Electrotechnical Commission, "Enterprise-control system integration — Part 1: Models and terminology," IEC 62264-1:2013, International Electrotechnical Commission, 2013. [Online]. Available: <https://www.iso.org/standard/57308.html>
- [6] DIN, "Reference architecture model Industrie 4.0 (RAMI 4.0)," DIN SPEC 91345:2016-04, DIN, Apr. 2016. [Online]. Available: <https://www.dinmedia.de/en/technical-rule/din-spec-91345/250940128>
- [7] International Electrotechnical Commission, "OPC unified architecture — Part 1: Overview and concepts," International Electrotechnical Commission, Tech. Rep. IEC TR 62541-1:2020, Nov. 2020. [Online]. Available: <https://webstore.iec.ch/en/publication/61109>
- [8] F. Cirillo, G. Solmaz, I. Berzanskytė, M. Bauer, L. Sanchez, and A. Skarmeta, "A standard-based open source IoT platform: FIWARE," *Comput. Netw.*, vol. 169, 2020, Art. no. 106980, doi: [10.1109/IOTM.0001.1800022](https://doi.org/10.1109/IOTM.0001.1800022).
- [9] P. Varga et al., "Making system of systems interoperable—the core components of the arrowhead framework," *J. Netw. Comput. Appl.*, vol. 81, pp. 85–95, 2017, doi: [10.1016/j.jnca.2016.08.028](https://doi.org/10.1016/j.jnca.2016.08.028).
- [10] W. O. W. Group, "OWL 2 web ontology language document overview," (second edition), W3C Recommendation, Dec. 2012. [Online]. Available: <https://www.w3.org/TR/owl2-overview/>
- [11] M. Sadeghi, A. Carenini, O. Corcho, M. Rossi, R. Santoro, and A. Vogelsang, "Interoperability of heterogeneous systems-of-systems: From requirements to a reference architecture," *J. Supercomput.*, vol. 80, no. 7, pp. 8954–8987, 2024, doi: [10.1007/s11227-023-05774-3](https://doi.org/10.1007/s11227-023-05774-3).
- [12] A. Schlemitz and V. Mezhuyev, "Approaches for data collection and process standardization in smart manufacturing: Systematic literature review," *J. Ind. Inf. Integration*, vol. 38, 2024, Art. no. 100578, doi: [10.1016/j.jii.2024.100578](https://doi.org/10.1016/j.jii.2024.100578).
- [13] L. Daniele, F. den Hartog, and J. Roes, "Created in close interaction with the industry: The smart appliances reference (SAREF) ontology," in *Proc. Int. Workshop Formal Ontologies Meet Industry (FOMI 2015)*, Springer, 2015, vol. 225, pp. 100–112.
- [14] R. van der Weerd, V. de Boer, L. Daniele, B. Nouwt, and R. Siebes, "Making heterogeneous smart home data interoperable with the SAREF ontology," *Int. J. Metadata, Semantics Ontol.*, vol. 15, no. 4, pp. 280–293, 2021, doi: [10.1504/IJMSO.2021.125893](https://doi.org/10.1504/IJMSO.2021.125893).
- [15] R. van der Weerd, V. de Boer, L. Daniele, and B. Nouwt, "Validating SAREF in a smart home environment," in *Proc. 14th Int. Conf. Metadata Semantic Res. (MISR 2020)* 2021, vol. 1355, pp. 35–46, doi: [10.1007/978-3-030-71903-6_4](https://doi.org/10.1007/978-3-030-71903-6_4).
- [16] M. Compton et al., "The SSN ontology of the W3C semantic sensor network incubator group," *Web Semantics: Science, Serv. Agents World Wide Web*, vol. 17, pp. 25–32, 2012, doi: [10.1016/j.websem.2012.05.003](https://doi.org/10.1016/j.websem.2012.05.003).
- [17] A. Haller, K. Janowicz, S. Cox, D. L. Phuoc, K. Taylor, and M. Lefrançois, "Semantic sensor network ontology," W3C Recommendation, 2017. [Online]. Available: <https://www.w3.org/TR/vocab-ssn/>
- [18] B. Smith et al., "A first-order logic formalization of the industrial ontology foundry signature using basic formal ontology," in *Proc. Joint Ontology Workshops (ser. CEUR Workshop Proceedings)* 2019, Accessed: Feb. 02, 2026. [Online]. Available: <https://philpapers.org/rec/SMAIFL-3>
- [19] M. Drobnjakovic, B. Kulvatunyou, F. Ameri, C. Will, B. Smith, and A. Jones, "The industrial ontologies foundry (IOF) core ontology," in *Proc. FOMI 2022: 12th Int. Workshop Formal Ontologies Meet Industry*, Tarbes, France, Sep. 2022, pp. 1–13, Accessed: Feb. 02, 2026. [Online]. Available: <https://philpapers.org/rec/DROTIOF>
- [20] R. Schiekofner, S. Grimm, M. M. Brandt, and M. Weyrich, "A formal mapping between OPC UA and the semantic web," in *Proc. IEEE 17th Int. Conf. Ind. Informat.*, 2019, vol. 1, pp. 33–40.

- [21] A. Perzylo, S. Profanter, M. Rickert, and A. Knoll, "OPC UA node-set ontologies as a pillar of representing semantic digital twins of manufacturing resources," in *Proc. 24th IEEE Int. Conf. Emerg. Technol. Factory Autom.*, Zaragoza, Spain, 2019, pp. 1085–1092, doi: [10.1109/ETFA.2019.8868954](https://doi.org/10.1109/ETFA.2019.8868954).
- [22] "OPC 10101: Binding for Web of Things (WoT)," OPC Foundation, Tech. Rep. 10101, Jan. 2026, released Jan. 08, 2026. [Online]. Available: <https://reference.opcfoundation.org/WoT/Binding/v100/docs/>
- [23] S. Kaebisch, M. McCool, and E. Korkan, "Web of things (WoT) thing description 1.1" World Wide Web Consortium (W3C), W3C Recommendation, Dec. 2023, published Dec. 05, 2023. [Online]. Available: <https://www.w3.org/TR/wot-thing-description11/>
- [24] B. Katti, C. Plociennik, M. Ruskowski, and M. Schweitzer, "Sa-OPC-UA: Introducing semantics to OPC-UA application methods," in *Proc. 14th IEEE Conf. Autom. Sci. Eng.*, Munich, Germany, IEEE, 2018, pp. 1189–1196, doi: [10.1109/COASE.2018.8560463](https://doi.org/10.1109/COASE.2018.8560463).
- [25] B. Katti, C. Plociennik, and M. Schweitzer, "SemOPC-UA: Introducing semantics to OPC-UA application specific methods," *IFAC-PapersOnLine*, vol. 51, no. 11, pp. 1230–1236, 2018, doi: [10.1016/j.ifacol.2018.08.343](https://doi.org/10.1016/j.ifacol.2018.08.343).
- [26] S. R. Bader and M. Maleshkova, "The semantic asset administration shell," in *Proc. Semantic Syst. Power AI Knowl. Graphs* (ser. Lecture Notes in Computer Science), 2019, pp. 159–174. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-030-33220-4_12
- [27] H. Knublauch and D. Kontokostas, "Shapes constraint language (SHACL)" W3C Recommendation, 2017. [Online]. Available: <https://www.w3.org/TR/shacl/>
- [28] ETSI, "Context information management (CIM); NGSI-LD API," ETSI, ETSI Group Specification ETSI GS CIM 009," Jul. 2025, v1.9.1 (2025-07). [Online]. Available: https://www.etsi.org/deliver/etsi_gs/CIM/001_099/009/01.09.01_60/gs_CIM009v010901p.pdf
- [29] V. Araujo, K. Mitra, S. Saguna, and C. Åhlund, "Performance evaluation of FIWARE: A cloud-based IoT platform for smart cities," *J. Parallel Distrib. Comput.*, vol. 132, pp. 250–261, 2019, doi: [10.1016/j.jpdc.2018.12.010](https://doi.org/10.1016/j.jpdc.2018.12.010).
- [30] A. Taherkordi, C. Henry, D. Ahlers, and M. Fischer, "Reshaping smart cities through NGSI-LD enrichment: A data pipeline and digital twin approach," *Sensors*, vol. 24, no. 6, 2024, Art. no. 1858, doi: [10.3390/s24061858](https://doi.org/10.3390/s24061858). [Online]. Available: <https://www.mdpi.com/1424-8220/24/6/1858>
- [31] Y.-S. Jeong, J.-W. Byun, Y.-S. Oh, and H.-C. Kang, "City data hub: A cloud-based IoT platform for smart cities," *Sensors*, vol. 20, no. 23, 2020, Art. no. 7000, doi: [10.3390/s20237000](https://doi.org/10.3390/s20237000). [Online]. Available: <https://www.mdpi.com/1424-8220/20/23/7000>
- [32] R. Dautov, S. Tverdal, A. S. Bondevik, S. A. Frøshaug, V. Szabo, and J. R. Fiksdal, "Data interoperability using smart data models and NGSI-LD for the norwegian agrifood sector," in *Proc. 20th Conf. Comput. Sci. Intell. Syst.*, 2025, pp. 297–302, Accessed: Feb. 02, 2026. [Online]. Available: <https://annals-csis.org/proceedings/2025/pliks/5425.pdf>
- [33] J. Delsing, *IoT Automation: Arrowhead Framework*. Boca Raton: CRC Press, 2017.
- [34] J. van Deventer, "A model based implementation of an IoT framework," in *Proc. IEEE Int. Symp. Syst. Eng.*, 2025, pp. 1–7, doi: [10.1109/ISSE65546.2025.11370084](https://doi.org/10.1109/ISSE65546.2025.11370084).
- [35] J. Cuenca, "Arrowhead system ontology," Jun. 2021, ontology specification draft. Accessed: Apr. 15, 2026. [Online]. Available: <https://javiercuenca.github.io/i40go.github.io/domain-task/arrowheadsystem/1.0/index-en.html>
- [36] "ontology Arrowhead," 2026, digital Reference subontology documentation, ontology specification draft, Accessed: Apr. 15, 2026. [Online]. Available: <https://ifx-dr.github.io/DigitalReference/subontologies/Arrowhead/index-en.html>
- [37] T. R. Gruber, "A translation approach to portable ontology specifications," *Knowl. Acquisition*, vol. 5, no. 2, pp. 199–220, 1993, doi: [10.1006/knac.1993.1008](https://doi.org/10.1006/knac.1993.1008). [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S1042814383710083>
- [38] S. Javed, C. Paniagua, I. Javed, J. van Deventer, and J. Delsing, "Runtime value chain analysis and cost accounting via microservices in agile manufacturing," *IEEE Open J. Ind. Electron. Soc.*, vol. 6, pp. 181–201, 2025, doi: [10.1109/OJIES.2025.3532664](https://doi.org/10.1109/OJIES.2025.3532664).
- [39] O. Wintercorn, J. Van Deventer, and C. Paniagua, "Unlocking the power of digital transformation: The role of ontologies," in *Proc. 2025 IEEE Int. Syst. Conf.*, 2025, pp. 1–6, doi: [10.1109/SysCon64521.2025.11014791](https://doi.org/10.1109/SysCon64521.2025.11014791).
- [40] IEC 81346-1:2022 Industrial systems, installations and equipment and industrial products — Structuring principles and reference designations — Part 1: Basic rules, IEC Std., 2022. [Online]. Available: <https://webstore.iec.ch/en/publication/64021>