Platform-Agnostic Resource and Application Profiles for Heterogeneous Cloud–Edge–Device Orchestration

Nektarios Deligiannakis*, Amr Mousa†, Vassilis Papataxiarhis*,
Michalis Loukeris*, Stathes Hadjiefthymiades*
*Department of Informatics and Telecommunications,
National and Kapodistrian University of Athens, 15784 Athens, Greece
Email: nekdel@di.uoa.gr, vpap@di.uoa.gr, mloukeris@di.uoa.gr, shadj@di.uoa.gr
†Virtual Vehicle Research GmbH, 8010 Graz, Austria
Email: amr.mousa@v2c2.at

Abstract—**Orchestration across cloud, edge, and device environments requires consistent and machine-readable representations for both resources and applications. This paper presents a platform-agnostic, schema-validated profiling approach that prioritizes deterministic validation and operational compatibility in control-plane critical paths. The core profile contract is platform-independent, while the current implementation uses Kubernetes primitives where native resource metadata is mapped through labels, annotations, and taints, and externally managed devices are represented through a `DeviceNode` Custom Resource synchronized by controller workflows. For applications, we define versioned profiles for native, device, and integration workloads using JSON Schema Draft-07. We operationalize these models through an Application Profile Manager (APM) that provides profile authoring support, schema validation, repository-backed lifecycle management, and OpenAPI-described programmatic access with ETag-based optimistic concurrency control. Evaluation is functional and correctness-oriented: it validates native/device life-cycle synchronization, schema-enforced profile rejection, repository consistency under concurrent updates, and an end-to-end placement workflow in the Kubernetes-based realization. The result is a practical modeling and governance foundation for continuum orchestration that can be extended to other orchestration substrates through adapter mappings.**

***Keywords*-Platform-Agnostic Modeling; Kubernetes; Cloud-Edge Continuum; Orchestration; Schema Validation; Application Profiles; Distributed Systems.**

I. INTRODUCTION

Automated orchestration across cloud, edge, and device environments requires a common representation for heterogeneous resources and application requirements. In practice, the control plane must combine expressiveness with deterministic validation, interoperability, and manageable lifecycle governance. To address this need, we present a platform-agnostic, schema-validated modeling approach for native nodes, device nodes, and application profiles.

The model is intentionally not tied to a single orchestrator. However, in this paper we implement and evaluate it using Kubernetes primitives because project deployment requirements mandate Kubernetes as the operational baseline [1]. The same schema contract can be mapped to other orchestration platforms through adapter-specific bindings.

The operational component of this approach is the Application Profile Manager (APM), which provides profile authoring support, schema validation, repository persistence, and API-based access. The APM includes three components: a command-line interface (`apmctl`), a versioned schema registry, and a repository service exposed through Representational State Transfer (REST) APIs.

This paper explicitly covers: (i) resource abstraction for native and device representations, (ii) application profile modeling for native/device/integration workloads, and (iii) governance and lifecycle semantics implemented by the APM. We do not claim a novel scheduling or optimization algorithm, and we do not cover compiler/backend workflows beyond profile modeling and management.

Our contributions are:

- **Platform-agnostic resource profile model**: A schema-governed resource description for native and device execution targets, with a Kubernetes mapping based on labels, annotations, taints, and Custom Resource Definitions (CRDs).
- **Device representation and synchronization workflow**: A controller-mediated lifecycle for externally managed devices, realized in this work through `DeviceNode` Custom Resources.
- **Application profiles and governance**: Versioned application profile schemas and an APM toolchain implementing validation, repository consistency controls, and API-based lifecycle management.

The remainder of this paper is structured as follows. Section II presents the background and design rationale. Section III describes the resource and application abstraction layer, covering the native node, device node representation, application profile taxonomy, and the APM. Section IV presents the evaluation, including executed scenarios and an end-to-end orchestration demonstration. Section V discusses the implications and limitations of the proposed approach. Section VI concludes the paper and outlines directions for future work.

II. BACKGROUND AND DESIGN RATIONALE

The design of a scalable continuum orchestration system depends on the representation used for resources and application requirements. A central decision is whether to place semantic reasoning in the control path or to use schema-validated descriptors with explicit attributes and predictable validation behavior.

Courtesy of IARIA Board and IARIA Press. Original source: ThinkMind Digital Library https://www.thinkmind.org

A. From Ontologies to Schema-Based Modeling

Ontology-based approaches such as the Web Ontology Language (OWL) provide formal semantics and inference capabilities, but their reasoning overhead and latency variance in tight control loops [2]–[6] motivate our use of schema-based validation instead.

B. Platform-Agnostic, Schema-Validated Models

We adopt a schema-based method using structured JavaScript Object Notation (JSON) [7] and YAML Ain't Markup Language (YAML) [8] documents with versioned validation contracts. The core model is platform-agnostic and can be mapped to different orchestration substrates through adapter layers.

Schema validation provides structural validity and version compatibility guarantees under declared schema versions. In this work, JSON Schema Draft-07 [9] is used as a pragmatic baseline because of mature tooling support and stable validator behavior [9][10]. The model intentionally avoids dynamic-reference-heavy features, prioritizing predictable validation in operational workflows.

This approach also reduces operational complexity compared to architectures that require dedicated semantic infrastructure (e.g., triple stores and reasoners). Profiles are validated with lightweight tools and stored in standard repositories, simplifying integration and maintenance.

C. Resource Models for a Heterogeneous Continuum

A unified model must cover both **native nodes** and **device nodes**. In the Kubernetes realization, native nodes correspond to worker nodes and device nodes are represented in-cluster through CRDs. Kubernetes extensibility mechanisms (labels, annotations, taints, CRDs) provide the required compatibility and incremental extensibility [11].

This representation supports continuum-specific attributes (e.g., accelerator capabilities, network characteristics, trust and energy indicators) while preserving compatibility with existing controllers and schedulers.

D. Application Models for Automated Orchestration

Automated deployment across heterogeneous environments requires machine-readable application profiles describing metadata, resource needs, Quality of Service (QoS) expectations, and deployment constraints. Structured profiles allow orchestration systems to match requirements to capabilities and perform repeatable lifecycle actions.

The APM operationalizes this model by validating profiles against versioned schemas, storing them in a repository, and exposing programmatic APIs.

E. Focused Related Work Positioning

KubeEdge targets device integration at the edge but does not define a portable schema contract for resources and applications across different orchestrators [12]. Node Feature Discovery exposes hardware and software capabilities as labels for scheduling, without a unified profile model [13]. OAM and KubeVela focus on application-centric abstractions for cloud-native workloads and only partially cover continuum-specific attributes such as trust, energy, and mobility [14][15]. Recent systems like QONNECT and REACH concentrate on optimization and control strategies and assume the existence of sufficiently rich descriptors [16][17]. In contrast, this paper contributes a platform-agnostic, schema-governed profiling and governance layer that unifies native and device node models and versioned application profiles, with a concrete but non-binding realization on Kubernetes.

III. RESOURCE AND APPLICATION ABSTRACTION LAYER

This section presents the proposed abstraction layer for resources and applications, covering its design challenges and resource and device representations.

A. Resource Abstraction Layer

The abstraction layer is designed under three main challenges: (i) representing heterogeneous native and device nodes under a single schema, (ii) exposing continuum-specific attributes without introducing ontology reasoning in the control path, and (iii) keeping the model portable while still mapping cleanly to Kubernetes primitives and governance mechanisms.

The resource abstraction distinguishes two categories of computational entities, illustrated in Figure 1:

- **Native Nodes**, corresponding to orchestrator-managed compute targets (implemented here as Kubernetes worker nodes);
- **Device Nodes**, representing computational resources that are externally managed and may not directly join the orchestrator cluster.

Figure 1. Node types: Native and Device nodes.

This distinction is platform-agnostic at the model level. In this paper, it is bound to Kubernetes-native mechanisms due to project deployment requirements.

B. Kubernetes Realization of the Native Node Model

The standard Kubernetes Node object consists of three primary components:

- `metadata` (ObjectMeta),
- `spec` (NodeSpec),
- `status` (NodeStatus).

These components capture scheduling policies, resource capacity (CPU, memory, storage), and runtime conditions. However, they do not natively represent continuum-relevant attributes such as energy efficiency, monetary cost classes, trust

Courtesy of IARIA Board and IARIA Press. Original source: ThinkMind Digital Library https://www.thinkmind.org

TABLE I. EXTENDED NATIVE NODE ATTRIBUTES

Category	Examples
Kubernetes Core	CPU, memory, storage, nodeInfo
Labels	NodeCategory, NodePool, geolocation, accelerator capacity (GPU, FPGA, TPU)
Taints	trustScore, securityLevel
Annotations	energyEfficiency, monetaryCost, bandwidth, latency, packetLoss

levels, geo-location, mobility characteristics, or device-specific metrics.

To address these limitations, the abstraction layer maps the platform-agnostic native node profile to Kubernetes labels, annotations, taints, and related objects. This approach enriches node metadata while remaining fully compatible with Kubernetes APIs and scheduler logic.

Table I summarizes the attribute categories introduced for Native Nodes.

This schema-driven extension preserves low-latency scheduling while enabling richer placement decisions across heterogeneous environments.

In this model, trust and security signals are encoded to support both filtering and exclusion semantics. Labels/annotations expose descriptive attributes for matching and scoring, while taints are used for explicit exclusion or guarded scheduling when policy requires stronger protection boundaries.

C. Device Node Representation via CRDs

Device nodes do not operate as orchestrator workers. In the Kubernetes implementation, they are represented within the control plane as Custom Resources following the standard `metadata/spec/status` structure.

Each `DeviceNode` resource captures static attributes (e.g., hardware type, CPU/GPU model, operating system) and dynamic runtime metrics (e.g., CPU utilization, memory usage, connectivity status, battery level). This design ensures that orchestration components can query and monitor device capabilities through native control-plane APIs.

The system uses controller-based methods to keep physical devices in sync with their associated Custom Resources. Devices send their registration and status information through Mosquitto Message Queuing Telemetry Transport (MQTT) [18]. The Kubernetes API server creates or updates a Custom Resource after the DeviceNode schema validation process.

If expected updates are missed beyond configured timeout bounds, the controller marks the device offline and temporarily removes it from the scheduling candidate pool. This reconciliation mechanism leverages Kubernetes' controller pattern [19] to maintain consistency between desired and observed state.

D. Design Properties

The Node Model abstraction satisfies the following properties:

- **Portability**: Platform-agnostic core schema with adapter-based platform mappings;
- **Compatibility**: Kubernetes-compatible realization using native primitives;
- **Extensibility**: Incremental addition of new attributes without schema disruption;
- **Determinism**: Schema-based validation avoids runtime reasoning overhead;
- **Scalability**: Event-driven updates support large fleets of heterogeneous nodes;
- **Low Latency**: No ontology reasoning in the scheduling critical path.

By modeling both Native and Device nodes through structured, schema-validated representations, the abstraction layer enables uniform resource discovery, monitoring, and scheduling across the computing continuum.

E. Application Profile Model

An Application Profile (AP) is a declarative YAML document that describes an application's operational requirements and intended deployment state. The model is designed to provide scalable, reproducible, and schema-validated application descriptions suitable for automated orchestration across heterogeneous environments.

Each profile follows a platform-neutral `metadata/spec/status` structure (with Kubernetes-inspired naming in our implementation):

- **metadata**: Identifying information such as name, namespace, labels, and annotations.
- **spec**: Desired state definition, including resource requirements, dependencies, Quality of Service (QoS) parameters, and deployment constraints.
- **status**: Observed runtime state, maintained by orchestration.

Profiles are versioned and validated against JSON Schema Draft-07 specifications, ensuring structural consistency and forward compatibility.

F. Profile Taxonomy

Three profile categories enable unified modeling across execution domains:

- **Native Profiles**: Target containerized or virtualized workloads deployed on orchestrator-managed native compute nodes (Kubernetes workers in this implementation).
- **Device Profiles**: Describe applications interacting with hardware-specific resources in edge and Internet of Things (IoT) environments.
- **Integration Profiles**: Define multi-component or composite services, including API endpoints, external dependencies, and orchestration policies.

Each category extends a shared base schema with type-specific attributes, ensuring modularity while preserving structural uniformity. Figures 2, 3 and 4 depict the three above profiles.

Courtesy of IARIA Board and IARIA Press. Original source: ThinkMind Digital Library https://www.thinkmind.org

Copyright (c) IARIA, 2026. ISBN: 978-1-68558-373-6

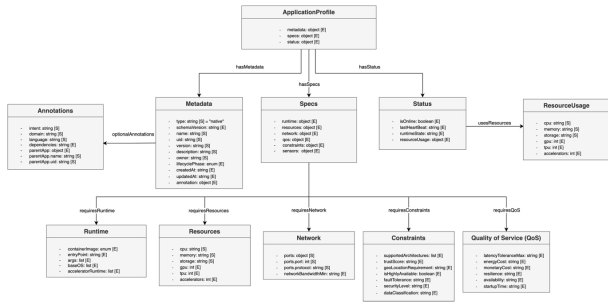

Figure 2. Data model diagram for a native application profile.

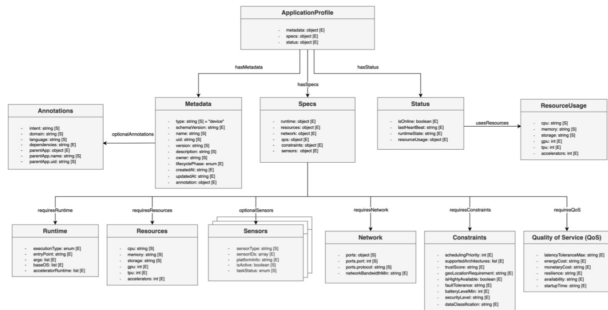

Figure 3. Data model diagram for a device application profile.

G. Schema Modularity and Attribute Scope

The schema design is modular and layered. A base schema defines common constructs (metadata, status, core identifiers), while type-specific schemas extend it with domain-relevant attributes.

The logical structure is organized into three primary components:

- **Core Definition**: Metadata, execution context, and QoS requirements.
- **Resource Requirements**: CPU, memory, accelerators, storage, and network specifications.
- **Deployment Constraints**: Scheduling preferences, geographical restrictions, trust parameters, and platform-specific requirements.

To clarify attribute origin, fields are categorized as:

- **Core**: Platform-agnostic attributes used across orchestrators;
- **Adapter-Specific**: Attributes mapped to a concrete platform realization (e.g., Kubernetes labels/taints/CRDs).

This separation preserves portability while enabling direct interoperability with Kubernetes-native tooling in the current implementation.

H. Application Profile Manager (APM)

The APM provides lifecycle management for application profiles, including authoring, validation, storage, and programmatic access, as illustrated in Figure 5.

The APM consists of three primary components:

- **Command-Line Interface (CLI) (`apmctl`)**: Enables template generation, schema validation, and repository interaction.

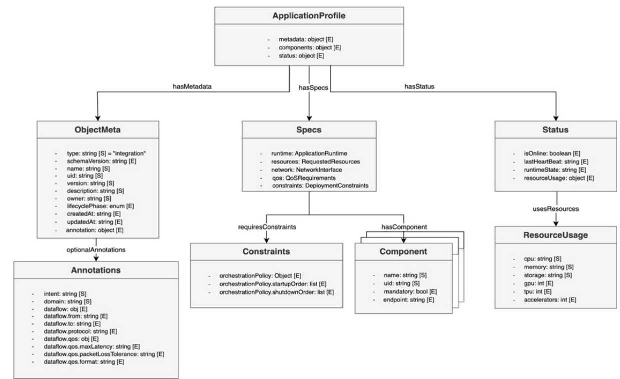

Figure 4. Data model diagram for an integration application profile.

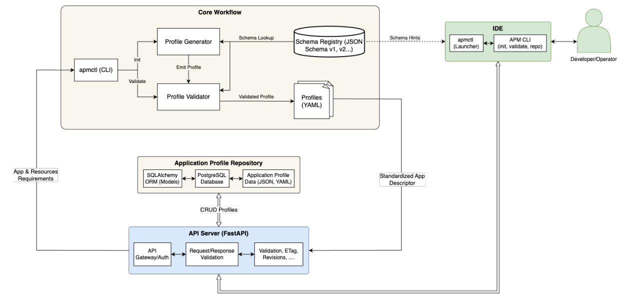

Figure 5. High-level architecture of the Application Profile Manager.

- **Schema Registry**: A versioned repository of JSON Schemas and corresponding YAML templates.
- **Profile Repository**: A persistent service exposing RESTful APIs for Create, Read, Update, Delete (CRUD) operations on profiles.

Validation is enforced using JSON Schema Draft-07 and automated through the `check-jsonschema` tooling. During validation, YAML profiles are converted to JSON and checked for required fields, data types, patterns, and enumerations according to the declared profile type and schema version.

The repository API is implemented using FastAPI [20] and automatically exports an OpenAPI 3 specification [21]. This provides synchronized interface documentation and standards-compliant programmatic access. The governance contract includes: (i) schema-version binding for each stored profile, (ii) explicit validation failure reporting for structural/type/constraint violations, and (iii) HTTP ETag-based optimistic concurrency control to prevent lost updates during concurrent writes.

The Application Abstraction Layer provides:

- **Deterministic Validation**: Schema-based enforcement eliminates runtime ambiguity;
- **Modularity**: Layered schema extensions support heterogeneous execution targets;
- **Interoperability**: Platform-agnostic schema contracts with a Kubernetes reference mapping for implementation compatibility;
- **Lifecycle Management**: Versioned profiles enable reproducible deployments and controlled evolution.

Courtesy of IARIA Board and IARIA Press. Original source: ThinkMind Digital Library https://www.thinkmind.org

By combining structured application descriptors with a schema-governed tool-chain, the abstraction layer enables automated matching between application requirements and resource capabilities across the computing continuum, while allowing extension to non-Kubernetes control environments through adapter mappings.

IV. EVALUATION

The proposed modeling layer was validated in heterogeneous environments, including local clusters, on-premises multi-node setups, and cloud-based infrastructure. The reported experiments correspond to the Kubernetes-based implementation selected for the project environment. The evaluation is functional and correctness-oriented: it verifies lifecycle synchronization, validation behavior, and repository consistency under representative scenarios.

*A. **RQ1: Does native node metadata synchronization remain correct under lifecycle events?***

Native nodes were instrumented through a DaemonSet-based synchronization loop. In each cycle, static attributes were read from the Kubernetes API, while dynamic metrics (e.g., network indicators, uptime, energy-related attributes) were recomputed and patched to the Node object using JSON patch operations.

Observed outcomes:

- Enriched metadata propagated to active worker nodes.
- Node join and node removal events were reflected without manual intervention.
- Dynamic attribute refresh did not disrupt scheduler operation during the test scenarios.
- When a metric source was unavailable, fallback behavior preserved profile consistency.

*B. **RQ2: Is device node registration and status reconciliation reliable?***

Device nodes were validated using an MQTT-based registration and synchronization workflow. Registration payloads were schema-checked before CR creation. Runtime metrics were patched to the `status` field through controller reconciliation.

Observed outcomes:

- Registration events produced valid `DeviceNode` resources with expected metadata/spec/status structure.
- Policy-based availability checks (e.g., battery-related constraints) were reflected in scheduling eligibility.
- Timeout-based liveness handling marked stale devices offline and removed them from candidate pools.
- Dynamic capability changes were propagated to the corresponding CR status.

Controller logs during the executed scenarios show successful API patch operations and consistent reconciliation behavior in our testbed.

*C. **RQ3: Do schema validation and repository controls enforce profile consistency?***

Application profiles were validated against JSON Schema Draft-07. Invalid inputs were rejected for structural violations, type mismatches, and enumeration errors. Versioned schemas constrained accepted structures by declared profile type/version.

The repository API enforced optimistic concurrency using HTTP ETags. Concurrent modification scenarios showed stale updates being rejected with precondition failures, preventing lost updates.

D. Executed scenarios and evidence

TABLE II. EXECUTED SCENARIOS AND OBSERVED OUTCOMES

Scenario	Expected	Observed	Evidence
Native metadata refresh	Node metadata enriched and updated after lifecycle events	Successful propagation and refresh in test runs	DaemonSet/controller logs and Node object state snapshots
Device registration	Valid registration creates/updates `DeviceNode` CR	CR created with expected fields	MQTT ingestion logs, controller events, CR snapshots
Device liveness timeout	Stale device marked unavailable/offline	Candidate removed after timeout handling	Controller logs and CR status transitions
Profile validation	Invalid profile rejected before persistence	Invalid inputs rejected by schema checks	Validation CLI/API responses
Concurrent profile updates	Stale write rejected to avoid lost update	Outdated ETag rejected (precondition failure)	Repository API request/response logs

E. End-to-end orchestration demonstration

An end-to-end scenario was executed in which an application profile specifying hardware type, operating system, battery threshold, and sensor requirements was submitted to the system. The scheduler successfully matched the application to a compatible Device Node and delegated execution. Upon device state changes (battery drop or network loss), the system automatically excluded the node and re-evaluated placement decisions.

These results show that the proposed layer provides deterministic schema validation, stable synchronization semantics, and repository consistency mechanisms suitable for continuum orchestration workflows. Large-scale throughput, tail-latency, and comparative baseline benchmarking are left for future work.

V. DISCUSSION

The proposed modeling layer represents a deliberate shift from ontology-centric runtime reasoning toward schema-validated representations. This decision reflects a trade-off between formal semantic expressiveness and operational predictability in control-plane workflows.

Ontology-driven approaches provide strong reasoning capabilities but can introduce additional overhead and latency variance as models and engines grow more complex. In distributed

Courtesy of IARIA Board and IARIA Press. Original source: ThinkMind Digital Library https://www.thinkmind.org

Copyright (c) IARIA, 2026. ISBN: 978-1-68558-373-6

environments with tight feedback loops, this variance is problematic, whereas schema-based validation performs structural checks before runtime placement, keeping the critical path simpler and more predictable.

Nevertheless, schema-based modeling imposes certain limitations. Unlike ontology-driven systems, it does not support automated semantic inference beyond explicitly defined attributes. Complex reasoning tasks must therefore be implemented in higher-level optimization components rather than embedded within the data representation layer. The architecture intentionally separates descriptive modeling from optimization logic to preserve performance and maintainability.

Overall, the abstraction layer demonstrates that structured, schema-governed models can provide sufficient expressiveness for continuum orchestration while maintaining deterministic behavior and scalability at runtime.

VI. Conclusions and Future Work

This paper presented a platform-agnostic, schema-governed resource and application profiling layer for cloud–edge–device orchestration. The approach combines platform-neutral profile contracts with a Kubernetes realization (native node enrichment and `DeviceNode` CRDs), together with versioned application profiles and an APM governance toolchain.

The evaluation focused on functional correctness in the Kubernetes-based implementation: synchronization behavior, schema validation outcomes, repository consistency, and end-to-end orchestration flow under representative scenarios. These results support the claim that the proposed layer is a practical and deterministic foundation for higher-level continuum orchestration components.

Future work will focus on three directions. First, quantitative scale and performance studies will measure control-plane overhead, update throughput, and tail latencies under stress. Second, security hardening will address identity, authorization, admission control, and multi-tenant boundaries. Third, integration with optimization/scheduling frameworks will evaluate how enriched descriptors affect placement quality and Service Level Objective (SLO) outcomes.

Acknowledgments

This work has been partially supported by the HYPER-AI project, funded by the European Commission under Grant Agreement 101135982 through the Horizon Europe research and innovation program (https://hyper-ai-project.eu/).

References

[1] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, omega, and kubernetes," *Communications of the ACM*, vol. 59, no. 5, pp. 50–57, 2016. DOI: 10.1145/2890784.

[2] V. Sazonau, U. Sattler, and G. Brown, "Predicting performance of OWL reasoners: Locally or globally?" In *Proceedings of the International Semantic Web Conference (ISWC)*, 2012, pp. 1–16.

[3] W. V. Woensel and S. S. R. Abidi, "Optimizing and benchmarking OWL2 RL for semantic reasoning on mobile platforms," *Journal of Web Semantics*, vol. 27–28, pp. 1–18, 2013.

[4] C. Seitz and R. Schönfelder, "Rule-based OWL reasoning for specific embedded devices," in *Proceedings of the International Semantic Web Conference (ISWC)*, 2011, pp. 237–252. DOI: 10.1007/978-3-642-25093-4_16.

[5] W. Tai, J. Keeney, and D. O'Sullivan, "COROR: A composable rule-entailment OWL reasoner for resource-constrained devices," in *Proceedings of the International Web Rule Symposium (RuleML)*, 2011, pp. 212–226. DOI: 10.1007/978-3-642-22546-8_17.

[6] A. I. Maarala, X. Su, and J. Riekki, "Semantic reasoning for context-aware internet of things applications," *IEEE Internet of Things Journal*, vol. 4, no. 2, pp. 461–473, 2017. DOI: 10.1109/JIOT.2016.2587060.

[7] T. Bray, *The JavaScript Object Notation (JSON) Data Interchange Format*, RFC 8259, [retrieved: April, 2026], 2017. [Online]. Available: https://www.rfc-editor.org/rfc/rfc8259.

[8] *YAML Ain't Markup Language (YAML) Version 1.2*, [retrieved: April, 2026], 2009. [Online]. Available: https://yaml.org/spec/1.2/spec.html.

[9] "JSON Schema Draft-07," [retrieved: April, 2026], 2025. [Online]. Available: https://json-schema.org/draft-07.

[10] "Check-jsonschema," [retrieved: April, 2026], 2025. [Online]. Available: https://github.com/python-jsonschema/check-jsonschema.

[11] The Kubernetes Authors, "Custom resource definitions," [retrieved: April, 2026], 2025. [Online]. Available: https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/.

[12] KubeEdge Authors, "Kubeedge documentation," [retrieved: April, 2026], 2026. [Online]. Available: https://kubeedge.io/docs/.

[13] Kubernetes SIG Node, "Node feature discovery," [retrieved: April, 2026], 2026. [Online]. Available: https://kubernetes-sigs.github.io/node-feature-discovery/.

[14] Open Application Model Community, "Open application model (oam) specification," [retrieved: April, 2026], 2026. [Online]. Available: https://oam.dev/.

[15] KubeVela Authors, "Kubevela documentation," [retrieved: April, 2026], 2026. [Online]. Available: https://kubevela.io/docs/.

[16] QONNECT Authors, "QONNECT: QoS-aware continuum orchestration with kubernetes-native control planes," [retrieved: April, 2026], 2025. [Online]. Available: https://arxiv.org/abs/2510.09851.

[17] REACH Authors, "REACH: Reinforcement learning-based rescheduling in heterogeneous kubernetes continuum environments," [retrieved: April, 2026], 2025. [Online]. Available: https://arxiv.org/abs/2510.06675.

[18] *MQTT version 3.1.1*, [retrieved: April, 2026], OASIS, 2014. [Online]. Available: http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/.

[19] The Kubernetes Authors, "Kubernetes controllers," [retrieved: April, 2026], 2025. [Online]. Available: https://kubernetes.io/docs/concepts/architecture/controller/.

[20] "Fastapi framework," [retrieved: April, 2026], 2025. [Online]. Available: https://fastapi.tiangolo.com/.

[21] "Openapi specification," [retrieved: April, 2026], 2025. [Online]. Available: https://github.com/OAI/OpenAPI-Specification.

Courtesy of IARIA Board and IARIA Press. Original source: ThinkMind Digital Library https://www.thinkmind.org

Copyright (c) IARIA, 2026. ISBN: 978-1-68558-373-6