FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Edge-Centric Service-Oriented Architecture for IEC 61499 Distributed Control

Tomás Cabral Torres

Mestrado em Engenharia Informática e Computação

Supervisor: Prof. Rui Pinto

July 15, 2026

Edge-Centric Service-Oriented Architecture for IEC 61499 Distributed Control

Tomás Cabral Torres

Mestrado em Engenharia Informática e Computação

Approved in oral examination by the committee:

President: Prof. Konstantinos Bletsas
Examiner: Prof. José Barbosa
Member: Prof. Rui Pinto

July 15, 2026

Resumo

A transformação industrial associada à Indústria 4.0 tem promovido a integração entre tecnologias de informação, sistemas de automação e processos físicos. Neste contexto, os Sistemas Ciber-Físicos de Produção (CPPS) permitem representar e coordenar componentes distribuídos em ambientes industriais. A norma IEC 61499 contribui para este objetivo através de um modelo baseado em Blocos Funcionais, que suporta modularidade, reutilização e distribuição da lógica de controlo. O Dynamic INtelligent Architecture for Software and MOdular REconfiguration (DINASORE) implementa este modelo num ambiente de execução em Python orientado para aplicações distribuídas e reconfiguráveis.

Apesar destas capacidades, a execução distribuída não fornece, por si só, mecanismos para representar persistentemente as instâncias de execução, descobrir funcionalidades já implantadas ou compor pipelines independentes. No fluxo tradicional do DINASORE, a estrutura das aplicações e os detalhes de comunicação têm de ser definidos explicitamente, o que aumenta a dependência de informação sobre endereços, interfaces e componentes específicos. A Arquitetura Orientada a Serviços (SOA) oferece princípios como abstração, baixo acoplamento, descoberta e composição, enquanto os Gémeos Digitais e o middleware permitem manter uma representação comum e persistente das entidades distribuídas.

Este trabalho propõe uma arquitetura orientada a serviços e centrada no edge para estender a orquestração de aplicações IEC 61499 executadas pelo DINASORE, preservando a execução local da lógica de controlo. A implementação do protótipo utiliza o Eclipse Ditto como camada de middleware para representar instâncias DINASORE, pipelines, serviços e bindings através de Gémeos Digitais. Cada instância é associada a um orquestrador responsável por traduzir operações do middleware em comandos de gestão IEC 61499. Os Blocos Funcionais implantados podem ser registados e descobertos como serviços, enquanto os bindings permitem estabelecer comunicação entre pipelines independentes. Um dashboard reúne a gestão de instâncias, a construção e implantação de pipelines, a monitorização, a descoberta de serviços e a configuração de bindings. O fluxo de trabalho proposto foi avaliado em termos de overhead de execução, usabilidade, esforço de engenharia e escalabilidade, mostrando melhor suporte à orquestração no cenário testado, mas introduzindo custos adicionais de recursos e comunicação.

Palavras-chave: Middleware • IEC 61499 • Arquitetura orientada a serviços • Sistemas Ciber-Físicos de Produção

Abstract

The industrial transformation associated with Industry 4.0 has promoted the integration of information technologies, automation systems, and physical processes. In this context, Cyber-Physical Production Systems (CPPS) enable distributed components to be represented and coordinated in industrial environments. IEC 61499 contributes to this objective through a Function Block-based model that supports modularity, reuse, and the distribution of control logic. The Dynamic INtelligent Architecture for Software and MOdular REconfiguration (DINASORE) implements this model through a Python-based Runtime Environment designed for distributed and reconfigurable applications.

Despite these capabilities, distributed execution does not by itself provide mechanisms for persistently representing runtime instances, discovering already deployed functionality, or composing independent pipelines. In the traditional DINASORE workflow, application structure and communication details must be defined explicitly, increasing the dependency on information about addresses, interfaces, and specific components. Service-Oriented Architecture (SOA) provides principles such as abstraction, loose coupling, discovery, and composition, while Digital Twins and middleware make it possible to maintain a common and persistent representation of distributed entities.

This work proposes an edge-centric service-oriented architecture for extending the orchestration of IEC 61499 applications executed by DINASORE while preserving the local execution of control logic. The prototype implementation uses Eclipse Ditto as a middleware layer to represent DINASORE instances, pipelines, services, and bindings through Digital Twins. Each runtime instance is associated with an orchestrator responsible for translating middleware operations into IEC 61499 management commands. Deployed Function Blocks can be registered and discovered as services, while bindings establish communication between independent pipelines. A dashboard integrates instance management, pipeline construction and deployment, monitoring, service discovery, and binding configuration. The proposed workflow was evaluated in terms of runtime overhead, usability, engineering effort, and scalability, showing improved orchestration support in the tested scenario while introducing additional resource and communication costs.

Keywords: Middleware • IEC 61499 • Service-oriented Architecture • Cyber-Physical Production Systems

The United Nations Sustainable Development Goals

The United Nations Sustainable Development Goals (SDGs) provide a global framework for addressing major social, economic, and environmental challenges. This dissertation relates mainly to SDG 9, which concerns industry, innovation, and infrastructure, and secondarily to SDG 8, which concerns productive work, technological upgrading, and sustainable economic growth.

The contribution of this work is indirect. The dissertation does not evaluate industrial sustainability, economic impact, or social outcomes directly. Instead, it contributes to the technological foundations that may support more flexible, observable, and maintainable industrial automation systems. By proposing an edge-centric service-oriented orchestration workflow for DINASORE-based IEC 61499 applications, the work addresses software-level mechanisms for representing runtime instances, deployed pipelines, services, and bindings in distributed control systems.

The specific Sustainable Development Goals considered in this dissertation are:

SDG 9 Build resilient infrastructure, promote inclusive and sustainable industrialization and foster innovation.

SDG 8 Promote sustained, inclusive and sustainable economic growth, full and productive employment and decent work for all.

SDG	Target	Contribution	Performance Indicators and Metrics
9	9.4	The proposed orchestration workflow may support the modernization of industrial software infrastructure by improving the visibility, configurability, and maintainability of distributed IEC 61499 applications.	Prototype validation through runtime representation, service discovery, binding creation, workflow-completion time, resource-overhead measurements, and scalability tests.
	9.5	The dissertation contributes to research and innovation in distributed industrial automation by combining IEC 61499, service-oriented principles, middleware, and edge-centric orchestration.	Research output, prototype implementation, experimental evaluation, and peer-reviewed publication related to service-oriented approaches for IEC 61499 distributed control systems.
8	8.2	The work may indirectly support engineering productivity by reducing manual configuration effort in the evaluated DINASORE workflow and by improving access to deployed functionality through service discovery and bindings.	Comparison between the traditional and proposed workflows, including configuration time, participant task-completion time, usability questionnaire results, and qualitative feedback.

These contributions should be interpreted cautiously. The dissertation evaluates a prototype in a controlled laboratory setup and does not directly measure industrial productivity, economic growth, environmental impact, or long-term sustainability. The SDG contribution is therefore technological and indirect, related to the potential of service-oriented orchestration to support more adaptable and maintainable industrial automation software.

Acknowledgements

A realização desta dissertação representa o culminar de uma etapa importante do meu percurso académico, a qual não teria sido possível sem o apoio, a orientação e o incentivo de várias pessoas e instituições. Assim, gostaria de expressar o meu sincero agradecimento a todos aqueles que, de forma direta ou indireta, contribuíram para a concretização deste trabalho. Em especial, gostaria de agradecer:

- **À minha família**, pelo apoio constante, pela confiança e pelo incentivo ao longo de todo o meu percurso académico. Aos meus pais, deixo um agradecimento especial por todo o esforço, compreensão e suporte que sempre me proporcionaram. Ao meu irmão, *Mateus Cabral*, agradeço igualmente pela presença, pelo apoio e pela motivação ao longo desta caminhada;
- **Ao Professor *Rui Pinto***, orientador desta dissertação, pela disponibilidade, orientação e acompanhamento prestados ao longo do desenvolvimento deste trabalho. Agradeço os esclarecimentos, as sugestões, a paciência e o incentivo constante, que foram fundamentais para a concretização desta dissertação;
- **À Faculdade de Engenharia da Universidade do Porto**, pela formação académica proporcionada e pelas oportunidades de aprendizagem e desenvolvimento pessoal e profissional que marcaram este percurso;
- **Aos colegas, docentes e amigos** que, de alguma forma, contribuíram para esta etapa, quer através do apoio, da partilha de conhecimento ou do incentivo nos momentos mais exigentes.

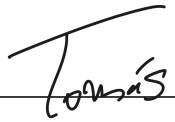
<Tomás Torres>

Declaração de Honra

Declaro que a presente dissertação é de minha autoria e não foi previamente apresentada e avaliada nesta ou em outra instituição. As referências a outros autores (afirmações, ideias, pensamentos), assim como a utilização de trabalhos publicados respeitam escrupulosamente as regras da atribuição, e encontram-se devidamente indicadas no texto e nas referências bibliográficas, de acordo corn as normas de referenciação. Tenho consciência de que a prática de plágio ou de autoplágio constituem ilícitos académicos, conforme estabelecido no Código de Ética da U.Porto.

Mais declaro que incluí texto(s) publicado(s) em coautoria, tendo especificado, com transparência e rigor, a colaboração de cada coautor e informado sobre a eventual existência de alguma outra dissertação em que também tenha sido incluído. São também incluídos a autorização da revista em que se encontra publicado, assim como a acordo expresso assinado por todos as coautores para a inclusão do artigo.

<Tomás Cabral Torres>



Contents

List of Figures

List of Tables

List of Acronyms

AMQP	Advanced Message Queuing Protocol
API	Application Programming Interface
CPU	Central Processing Unit
CPPS	Cyber-Physical Production System
CPS	Cyber-Physical System
DINASORE	Dynamic INtelligent Architecture for Software and MOdular REconfiguration
ESB	Enterprise Service Bus
FB	Function Block
HERE	Hybrid Synchronous–Asynchronous Runtime Environment
HTTP	Hypertext Transfer Protocol
IDE	Integrated Development Environment
IoT	Internet of Things
IP	Internet Protocol
JSON	JavaScript Object Notation
MQTT	Message Queuing Telemetry Transport
OPC UA	Open Platform Communications Unified Architecture
PLC	Programmable Logic Controller
RAM	Random Access Memory
REST	Representational State Transfer
RTE	Runtime Environment
SDG	Sustainable Development Goal
SOA	Service-Oriented Architecture
SOMA	Service-Oriented Manufacturing Architecture
TCP	Transmission Control Protocol
XML	Extensible Markup Language

Chapter 1

Introduction

1.1 Context

Industry 4.0 describes the continuing transformation of industrial production through the integration of automation, communication, and information technologies. Technologies such as the Internet of Things (IoT), Cyber-Physical System (CPS), Edge Computing, and data analytics allow physical processes, software components, and human operators to exchange information and coordinate their actions [40]. Figure 1.1 summarizes some of the technological concepts associated with this transformation.

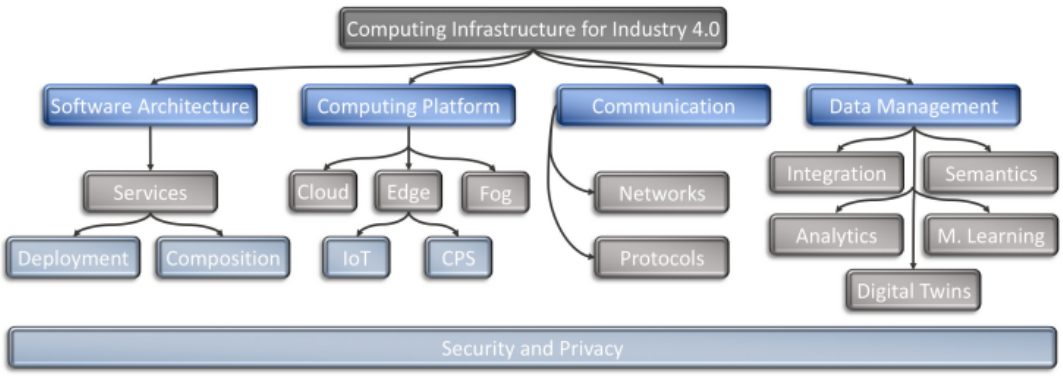


Figure 1.1: Main technological concepts associated with Industry 4.0 and their relationships [40].

In manufacturing environments, this integration is commonly represented through Cyber-Physical Production System (CPPS). A CPPS connects computational components with sensors, actuators, machines, and production processes, allowing information from the physical environment to be used for monitoring, decision-making, and control [29, 40]. Since these systems are distributed across heterogeneous devices, their operation depends not only on the execution of control logic, but also on the ability to coordinate components and adapt the system when its configuration or operational requirements change.

The edge layer has a particularly important role in this context. Devices located close to the physical process execute functions that may be sensitive to latency, network availability, and resource constraints [21]. For this reason, control logic is commonly kept close to the equipment, while higher-level mechanisms support management, coordination, and integration. The scope of this dissertation is therefore edge-centric: the proposed work preserves local control execution on edge devices and focuses on improving how distributed runtime instances are managed and connected.

IEC 61499 provides a component-oriented model for developing distributed and event-driven control applications. Its Function Block (FB) abstraction supports modularity, reuse, and the distribution of application logic across multiple devices [5, 46]. Runtime Environments (RTEs) execute these applications and provide the mechanisms required to deploy, start, monitor, and reconfigure their components. One such environment is the Dynamic INtelligent Architecture for Software and MOdular REconfiguration (DINASORE), a Python-based RTE designed for distributed and reconfigurable CPPS applications [16, 33].

Although IEC 61499 and DINASORE provide a suitable basis for distributed control, the management of multiple runtime instances and pipelines remains challenging. Runtime topology, available functionality, communication endpoints, and application connections are often defined explicitly, which makes system evolution and reconfiguration increasingly dependent on manual engineering decisions. Service-Oriented Architecture (SOA) provides principles such as service abstraction, loose coupling, discovery, and composition that can be used to address these limitations [10, 40]. This dissertation investigates how these principles can be incorporated into an IEC 61499 RTE workflow while preserving the execution responsibilities of the runtime.

1.2 Motivation

DINASORE supports the execution and reconfiguration of distributed IEC 61499 pipelines [16, 33], but its traditional workflow still requires the application structure and communication details to be defined explicitly. The engineer must know which runtime hosts each part of the application, which FBs are available, and how independent pipelines communicate. Communication between runtime instances commonly requires protocol-specific FBs and the manual configuration of addresses, ports, topics, and interfaces. As the number of devices and pipelines grows, this increases engineering effort and makes existing functionality more difficult to locate and reuse.

SOA offers a possible way to reduce this dependency on infrastructure details. By representing runtime capabilities as services, functionality can be described through explicit interfaces, registered, discovered, and composed without requiring the consumer to know its internal implementation [10, 12, 40]. Applying these principles to IEC 61499, however, requires more than adding a communication protocol. The architecture must represent runtime instances and deployed pipelines, maintain service descriptions, translate management operations into actions supported by the RTE, and coordinate communication between independently executed applications.

The motivation for this work is therefore to investigate how the DINASORE workflow can be supported by service-oriented mechanisms without changing the role of DINASORE as the runtime responsible for IEC 61499 execution. Rather than requiring engineers to manually identify runtime locations, available functionality, and communication details, a service-oriented workflow can make distributed functionality easier to describe, discover, and compose [10, 40]. This is particularly relevant when independently deployed pipelines need to interact or when existing functionality should be reused in a different application context.

Such an extension also raises design challenges. The orchestration mechanisms must remain compatible with existing runtime execution, support distributed deployment and reconfiguration, and remain suitable for resource-constrained edge environments. The motivation is therefore not only to improve abstraction and engineering workflow, but also to understand the trade-offs introduced when service-oriented coordination is added around an IEC 61499 runtime.

1.3 Problem

IEC 61499 supports distributed and event-driven control applications, but distributed execution alone does not provide a service-oriented orchestration model. In an IEC 61499 system, RTEs execute FB networks and may expose management operations for deployment, monitoring, and reconfiguration. However, these mechanisms do not necessarily provide a persistent representation of runtime instances, deployed pipelines, available capabilities, service descriptors, or bindings between independently deployed applications.

This limitation becomes relevant when distributed automation systems need to evolve after deployment. Reusing an FB already deployed in another pipeline, adding a new runtime instance, or connecting independently deployed applications requires knowledge of runtime locations, FB interfaces, network endpoints, and communication mechanisms. As a result, application functionality can remain coupled to infrastructure details, while discovery, composition, and reconfiguration depend on manual engineering decisions.

DINASORE is used in this dissertation as the concrete RTE case study through which this problem is implemented and evaluated. In the baseline DINASORE workflow, application deployment and reconfiguration depend on externally defined configurations, and deployed FB instances are not exposed through a persistent service registry. Runtime instances also do not share a common representation of their available capabilities, deployed pipelines, current service state, or bindings. These characteristics make DINASORE a suitable case for studying how service-oriented orchestration can be introduced around an existing IEC 61499 RTE without replacing its local execution model.

Existing IEC 61499 RTEs, service-oriented approaches, and middleware platforms address different parts of this problem, but they do not jointly provide the workflow required here: runtime and service discovery, persistent representation of pipelines, abstraction of deployed FB instances as services, and managed communication between independent IEC 61499 pipelines. The research problem is therefore the absence of an integrated service-oriented orchestration layer for

IEC 61499 RTEs that can support persistent runtime representation, service discovery, service composition, and inter-pipeline communication while preserving local FB execution.

1.4 Objectives

The main objective of this work is to design and evaluate an edge-centric service-oriented orchestration architecture for IEC 61499 applications executed by distributed RTEs. The architecture is defined at the level of roles, information structures, interaction flows, and trade-offs, so that it can be reasoned about beyond a single implementation. DINASORE is used as the prototype RTE and evaluation case study, allowing the proposed architecture to be instantiated and tested in a concrete distributed control environment.

Based on the problem presented in Section 1.3, the work pursues the following objectives:

- define an architecture that integrates SOA principles with IEC 61499 runtime orchestration while preserving local FB execution inside the RTE;
- define a persistent information model for representing runtime instances, deployed pipelines, runtime capabilities, service descriptors, and bindings;
- specify how deployed FB instances can be abstracted, registered, and discovered as services;
- specify how service composition and inter-pipeline communication can be supported through explicit binding mechanisms;
- instantiate the architecture in a DINASORE-based prototype using middleware support, runtime-associated orchestrators, service and binding mechanisms, and a dashboard interface;
- evaluate the practical benefits, engineering effort, runtime overhead, usability, and scalability limits of the implemented workflow in comparison with the traditional DINASORE workflow.

The objective is not to replace the IEC 61499 RTE or to redefine FB execution semantics. Instead, the work examines whether a service-oriented orchestration layer can improve abstraction, discovery, composition, and workflow integration around distributed IEC 61499 runtime instances, and which computational, communication, and engineering costs are introduced by that additional layer.

1.4.1 Research Questions

The work is guided by the following research questions:

RQ1: How can service-oriented principles be integrated into an orchestration layer around IEC 61499 runtime instances while preserving local FB execution?

RQ2: What architectural mechanisms are required to support persistent runtime representation, service discovery, service composition, and inter-pipeline communication in IEC 61499-based distributed control systems?

RQ3: What practical benefits, costs, and limitations emerge when the proposed service-oriented orchestration architecture is instantiated and evaluated through a DINASORE-based workflow?

1.4.2 Contributions

The main contributions of this dissertation are:

1. **An edge-centric service-oriented orchestration architecture for IEC 61499 distributed control systems.** The architecture separates local FB execution from orchestration concerns and defines how runtime instances, deployed pipelines, services, and bindings can be coordinated through an external service-oriented layer.

2. **A persistent information model for service-oriented IEC 61499 orchestration.** The model represents runtime instances, deployed pipelines, runtime capabilities, service descriptors, and bindings as persistent orchestration entities, allowing distributed applications to be discovered, inspected, and composed beyond the local runtime process.

3. **A service-abstraction and discovery mechanism for deployed FB instances.** The work defines how deployed FB instances can be represented as services, exposing their relevant inputs, outputs, events, runtime location, and availability while preserving the execution responsibilities of the IEC 61499 RTE.

4. **A binding model for inter-pipeline communication.** The proposed binding mechanism represents communication between FB instances hosted by independently deployed pipelines, allowing producer outputs to be connected to consumer inputs and trigger events without embedding all communication details directly in each application structure.

5. **A prototype implementation based on DINASORE and middleware-supported orchestration.** The architecture is instantiated through a prototype using DINASORE, Eclipse Ditto, Eclipse Mosquitto, runtime-associated orchestrators, service and binding mechanisms, and a dashboard interface.

6. **An experimental evaluation of the proposed workflow.** The implemented workflow is evaluated against the traditional DINASORE workflow in terms of runtime overhead, engineering effort, usability, and scalability, identifying both the practical benefits and the costs introduced by the service-oriented orchestration layer.

7. **An analysis of applicability, trade-offs, and limitations.** The dissertation discusses when the architecture is appropriate, when it is less suitable, which benefits it provides, which costs it introduces, and which limitations remain for future work.

1.5 Dissertation Structure

The remainder of this dissertation is organized as follows.

Chapter 2 presents the background and literature review required to understand the dissertation. It introduces CPPS design and implementation with IEC 61499, SOA in industrial automation, and middleware and DT platforms, before comparing representative approaches and identifying the research gap addressed by the work.

Chapter 3 presents the proposed architecture independently of the specific technologies used in the prototype. It defines the architectural scope and requirements of the work and describes the reference orchestration architecture, information model, interaction flows, and architectural trade-offs.

Chapter 4 describes the prototype implementation. It details the middleware deployment and information model, runtime integration, orchestrator implementation, service registration and discovery, binding mechanisms, DINASORE adaptations, and dashboard implementation.

Chapter 5 presents the experimental methodology and evaluates the prototype in terms of runtime overhead, usability and workflow, and scalability. The results are compared with the traditional DINASORE workflow and analysed according to the evaluation scenarios and metrics.

Chapter 6 discusses the main findings, interprets the evaluation results, answers the research questions, and identifies the limitations of the work.

Chapter 7 presents the final conclusions, summarizes the main contributions of the dissertation, and outlines future directions without treating them as a separate subchapter.

Chapter 2

Background and Literature Review

This chapter establishes the conceptual and technical basis required to understand the proposed work. As discussed in Section 1.3, the challenge is not limited to executing IEC 61499 applications, but also concerns how distributed runtime instances, deployed pipelines, services, and bindings can be represented, discovered, and orchestrated in an edge-centric environment. IEC 61499 provides the execution model, SOA provides principles for abstraction, discovery, and composition, and middleware and DT platforms provide mechanisms for persistence and integration. By relating these areas, the chapter clarifies what existing work already supports and where the gap addressed by the proposed architecture remains.

2.1 CPPS Design and Implementation with IEC 61499

As introduced in Chapter 1, CPPS combine computational, communication, and physical production elements in order to support monitoring, control, adaptation, and coordination in industrial environments. Compared with conventional production systems, CPPS place greater emphasis on connectivity, flexibility, remote access, and the integration of digital and physical capabilities [29, 42]. These characteristics are aligned with Industry 4.0 requirements, where production systems are expected to integrate heterogeneous devices, software services, sensors, actuators, and information systems while remaining adaptable to changes in configuration or production requirements [30, 31].

The edge layer is particularly relevant in this context because not all industrial functionality can be moved to remote cloud infrastructures. Control and monitoring functions may be sensitive to latency, availability, and local resource constraints, while higher-level services may still be needed for supervision, configuration, data aggregation, or coordination [21]. For this reason, this dissertation adopts an edge-centric perspective: the execution of control logic remains local to runtime instances, while the proposed architecture adds mechanisms for distributed orchestration, service discovery, and composition around those runtimes.

IEC 61499 is one of the main standards proposed for the design of distributed industrial automation applications. It defines a component-oriented and event-driven model in which application logic is represented as a network of interconnected FBs [27, 46]. Each FB encapsulates behavior behind an interface composed of event and data inputs and outputs. Events coordinate execution, while data connections carry the information exchanged between FB instances. This separation between event flow and data flow differentiates IEC 61499 from more traditional scan-based Programmable Logic Controller (PLC) programming models and supports the representation of distributed applications at system level [5, 46].

The standard also introduces concepts such as applications, devices, resources, FB types, FB instances, connections, and management commands. An IEC 61499 application can be designed independently of its final deployment topology and later mapped to resources running on different devices. This allows engineering tools to represent a distributed control application as a logical FB network, while runtime environments execute the mapped FB instances and handle their lifecycle. These characteristics make IEC 61499 relevant for modularity, portability, reconfiguration, and distribution in CPPS-oriented applications [5, 27].

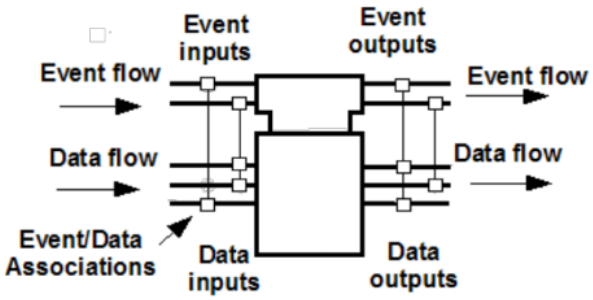


Figure 2.1: FB model in the IEC 61499 standard [7].

Figure 2.1 illustrates the basic IEC 61499 FB abstraction. The figure is important because it shows that an FB is not only a code unit, but also an interaction boundary: its behavior is accessed through explicit event and data interfaces. This property is later used by service-oriented approaches, which interpret FB interfaces as possible service contracts or message boundaries.

An IEC 61499 RTE is responsible for executing the deployed FB application. It creates FB instances, manages their lifecycle, schedules their execution, propagates events and data, and may expose management operations to create, modify, start, inspect, or remove application elements during execution. However, IEC 61499 does not prescribe a single runtime implementation strategy. Runtime implementations may therefore differ in event scheduling, buffering, multitasking, and concurrency behavior [19, 35]. Ferrarini and Veber [19] classify IEC 61499 execution approaches according to characteristics such as scan order and multitasking, showing that different combinations can lead to different runtime behavior.

Several IEC 61499 tools and runtimes have been proposed in both research and industrial contexts. Eclipse 4diac is one of the most widely used open-source frameworks for IEC 61499 engineering. It provides the 4diac Integrated Development Environment (IDE) for designing FB

applications and the FORTE runtime for executing them on target devices [18, 41]. The IDE supports the design, mapping, deployment, and monitoring of distributed FB networks, while the runtime executes the deployed application. This separation between engineering environment and runtime is important because it enables different IEC 61499 execution platforms to be used while preserving a common modeling workflow.

Eclipse 4diac. The Eclipse 4diac IDE provides the engineering environment used to create FB types and applications, configure devices and resources, map application elements to target devices, and deploy the resulting configuration. It can therefore be used to create and manage distributed FB pipelines and to send the corresponding management commands to compatible IEC 61499 runtimes [18]. The IDE also provides mechanisms for observing and interacting with deployed applications, including WATCH operations for variables and events and trigger operations for input events.

FORTE is the runtime most commonly associated with Eclipse 4diac. It is a portable C++ implementation of an IEC 61499 RTE that runs on the target device, receives the application configuration produced by the engineering environment, instantiates the FB network, and executes its event and data connections [18]. Although FORTE is frequently used together with 4diac IDE, the engineering environment is not limited to a single runtime. This separation allows IEC 61499 applications to be designed through a common workflow while being executed by different compatible RTEs.

DINASORE. DINASORE is a Python-based IEC 61499 RTE designed for distributed and reconfigurable cyber-physical applications [16, 33]. It executes applications as pipelines of interconnected FB instances and supports the integration of sensing, data processing, and control functionality within the same application structure. DINASORE separates the IEC 61499 description of an FB type from its executable Python implementation: the Extensible Markup Language (XML) definition describes the FB interface, while the Python component implements the internal behavior. This separation preserves compatibility with the IEC 61499 application model while allowing FB logic to use Python libraries for data processing, numerical computation, and machine-learning-oriented functionality.

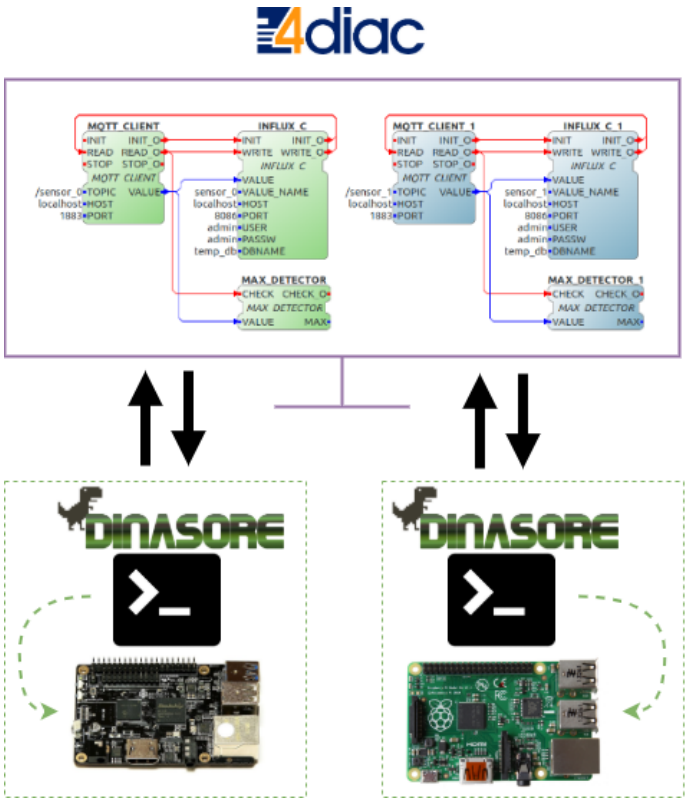


Figure 2.2: Overview of the DINASORE architecture, illustrating the interaction between IEC 61499 engineering, DINASORE runtime instances and Eclipse 4diac. Adapted from [16].

Figure 2.2 illustrates the relationship between Eclipse 4diac and the DINASORE framework. IEC 61499 applications are developed using Eclipse 4diac and subsequently deployed to DINASORE runtime instances executing on edge devices. This integration allows applications modeled according to the IEC 61499 standard to be executed in a distributed environment while preserving the Function Block execution model provided by DINASORE.

Existing literature shows that IEC 61499 has matured significantly as a basis for distributed and intelligent automation. Vyatkin [46] presents IEC 61499 as an enabler of decentralized automation intelligence, emphasizing the distribution of software components across networked devices and the potential for reconfigurable manufacturing systems. Lyu and Brennan [27] review IEC 61499 research from the perspective of distributed intelligent automation, identifying work on migration from IEC 61131-3, integration with enabling technologies, and the development of engineering environments. More recent reviews highlight that IEC 61499 research has moved beyond conceptual feasibility toward integration with Industry 4.0, intelligent automation, edge-oriented infrastructures, and industrial adoption barriers [5, 20].

At the same time, the literature also shows that adoption and implementation remain challenging. Industrial adoption is affected by migration from IEC 61131-3, training requirements, tool maturity, compatibility with existing systems, cybersecurity, and validation under realistic industrial conditions [3, 20, 27]. Ashiwal et al. [3] show that migration from IEC 61131-3 to IEC 61499 is not a simple one-to-one conversion, since the transition from cyclic to event-based execution

and the treatment of global variables may lead to syntactically valid but semantically inadequate applications. Bencherki et al. [5] also emphasize that many IEC 61499 studies remain prototype-oriented and that broader industrial validation is still limited. These limitations are relevant to this dissertation because they show that supporting distributed execution is not sufficient by itself: practical engineering workflows also require mechanisms for representing deployed applications, locating available functionality, coordinating distributed components, and managing runtime evolution.

Among the reviewed IEC 61499 RTEs, the Hybrid Synchronous–Asynchronous Runtime Environment (HERE) proposed by Zhou and Li [49] is relevant because it focuses on hybrid synchronous–asynchronous execution and runtime reconfiguration in edge-computing automation tasks.

Table 2.1: Comparison of representative IEC 61499 execution environments and integration frameworks.

Work	Distributed execution	Dynamic reconfiguration	Legacy integration	Service discovery and composition
FORTE [50]	Yes	Yes	No	No
HERE [49]	Yes	Partial	No	No
Gsellmann et al. [23]	Yes	Not the main focus	Yes	No
DINASORE [33]	Yes	Yes	No	No

Table 2.1 shows that distributed execution is a common capability across the reviewed frameworks, although their objectives differ considerably. FORTE focuses on portable IEC 61499 execution and runtime reconfiguration, whereas HERE proposes a hybrid synchronous–asynchronous IEC 61499 runtime for edge-computing automation tasks and supports reconfigurable FB networks and FB-type deployment through a just-in-time-based mechanism [49]. However, its support for dynamic FB-type reconfiguration is still partial, since consistency and integrity during dynamic type reconfiguration are not fully addressed. Gsellmann et al. [23] address the coexistence of cyclic IEC 61131-3 programs and event-driven IEC 61499 applications within a common engineering and execution framework. In contrast, DINASORE extends IEC 61499 execution toward reconfigurable cyber-physical applications through Python-based FBs, machine-learning libraries, and Open Platform Communications Unified Architecture (OPC UA) connectivity. Despite these advances, explicit support for service discovery, service composition, and orchestration remains outside the main scope of the reviewed approaches.

The reviewed IEC 61499 runtimes and engineering environments provide important capabilities for modular control, distributed execution, and reconfiguration. However, their primary concern remains the design and execution of FB applications. They do not, by themselves, provide a persistent service-oriented orchestration model for runtime instances, deployed pipelines, available services, and bindings between independently deployed applications. This limitation highlights the need for architectural principles that describe runtime capabilities as discoverable and composable services, rather than only as deployed elements inside a control application.

2.2 SOA in Industrial Automation

SOA is an architectural style in which software capabilities are organized and exposed as independent services. Each service provides a specific functionality through an explicit interface, allowing consumers to use that functionality without depending on its internal implementation. Communication between services is performed through defined messages and contracts, which reduces direct dependencies between distributed components [12, 40].

SOA does not prescribe a single implementation technology or communication protocol. Instead, it defines a set of architectural principles that can be implemented through different platforms and interaction mechanisms. By decomposing a complex system into smaller and self-contained services, the architecture can improve modularity, reuse, maintainability, and integration across heterogeneous software environments [40]. These characteristics are relevant to industrial automation, where applications frequently involve devices, control systems, and software components developed with different technologies. Service-oriented interfaces can provide a common interaction boundary between these components, supporting interoperability and allowing functionality to be reused or combined without exposing the internal implementation of each system [12].

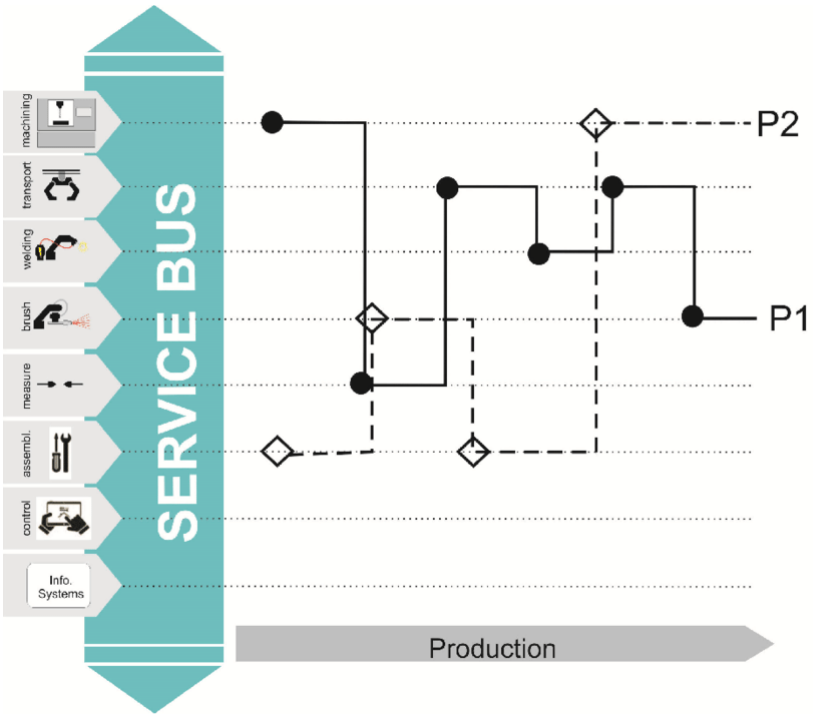


Figure 2.3: SOMA, illustrating manufacturing resources exposing their capabilities as services interconnected through a service bus. Adapted from Reis and Gonçalves [38].

Figure 2.3 illustrates the concept of a Service-Oriented Manufacturing Architecture (SOMA), in which manufacturing resources expose their capabilities as independent services connected through a common service bus. Rather than relying on fixed interactions between machines and

applications, production processes invoke the required services according to their current needs. This service-oriented organization promotes modularity, interoperability, and flexible composition of manufacturing capabilities, demonstrating the applicability of SOA concepts to industrial automation environments [38].

SOA principles. Dai, Vyatkin, and Christensen [12] identify a set of principles that characterize service-oriented systems and describe the relationships between them. These principles include reusability, service contracts, loose coupling, abstraction, composability, autonomy, statelessness, and discoverability. Reusability means that a service encapsulates functionality that can be used by different consumers and in different application contexts. A service contract defines the identity, interface, operations, exchanged messages, and relevant usage conditions of the service. Loose coupling reduces direct dependencies between service providers and consumers, while abstraction hides the internal implementation behind the exposed interface. Composability allows services to be combined into more complex processes, autonomy gives services control over their own logic and resources, statelessness reduces unnecessary dependence on previous interactions, and discoverability allows services to be located through metadata, registries, or repositories.

In the automation domain, SOA has been investigated as a way to address interoperability, flexibility, plug-and-play integration, and the integration of device-level functionality with higher-level systems. Thramboulidis [43] reviews the application of SOA in industrial automation and discusses the relevance of IEC 61499 as a possible basis for service-oriented automation models. Earlier research projects and approaches, such as SIRENA, SODA, SOCRADES, DPWS-based solutions, and semantic web service approaches, explored how service-based technologies could be applied to embedded devices, industrial components, and enterprise integration [43]. These works helped establish the relevance of SOA for industrial environments, but they did not fully solve how service principles should be integrated with the orchestration of IEC 61499 runtime capabilities and deployed control applications.

A key line of work directly connects SOA with IEC 61499. Dai, Vyatkin, and Christensen [12] compare SOA principles with the characteristics of IEC 61499 and argue that the IEC 61499 FB model provides a suitable basis for applying SOA to distributed automation systems. Their work highlights the connection between services and FBs: FBs provide explicit interfaces, communicate through event and data connections, and can be composed into networks. These properties make the IEC 61499 model suitable for representing service relationships in automation applications.

Dai et al. [10] later formalize the mapping between SOA and IEC 61499 and propose a service-based IEC 61499 execution environment architecture. In that mapping, services can be associated with FB types, service requests and responses can be related to event and data interfaces, and a service repository can be associated with runtime management. The proposed runtime architecture uses IEC 61499 management commands such as `CREATE`, `DELETE`, `START`, `READ`, and `WRITE` as part of a service-oriented execution environment. This work is particularly relevant here because it shows that IEC 61499 management mechanisms can be interpreted from a service-oriented perspective. The workflow considered here differs from that direction: instead of replacing the run-

time with a new SOA-based execution environment, it adds an orchestration layer around existing DINASORE runtime instances.

Further studies explored the use of IEC 61499 FB networks as a representation for service-oriented automation systems. Dai et al. [13] identify a limitation of SOA in automation: without a system-level representation, it can be difficult to understand and validate the relationships between services during the system lifecycle. They argue that IEC 61499 FB networks can provide a graphical and structural representation of services and their interactions. This observation is important for the present work because the proposed dashboard also relies on graph-like representations of pipelines, services, and bindings. Nevertheless, the objective here is not only to visually represent service interactions, but also to maintain persistent runtime state and enable discovery and binding between independently deployed pipelines.

SOA has also been applied to process automation software design. Dai et al. [11] propose a multi-layered service-oriented approach for designing distributed control software with IEC 61499. In their comparison between object-oriented programming, component-based design, and SOA, they emphasize that SOA adds loose coupling, service contracts, repositories, and orchestration/-composition mechanisms. Their case study demonstrates how a process control application can be structured according to service-oriented principles. This line of work supports the argument that SOA is not merely a communication technology, but a design paradigm for organizing reusable and composable automation functionality.

The relation between SOA and plug-and-play industrial cyber-physical systems has also been investigated. Dai, Huang, and Vyatkin [9] propose a plug-and-play service component approach based on WSDL, DPWS, and IEC 61499 FBs, where software components expose service interfaces that can be discovered and accessed by other distributed nodes. Jhunjhunwala, Atmojo, and Vyatkin [26] apply related ideas to smart sensor services in IEC 61499-based process automation, using standardized services and adapter connections to integrate sensor functionality into distributed automation applications. These works are relevant because they show how device-level or sensor-level functionality can be exposed as services and integrated into IEC 61499 systems. However, they focus on specific service-enabled components rather than a persistent management model for all runtime instances, pipelines, services, and bindings.

More recent work has also explored microservice-oriented approaches in industrial automation. Microservices share some principles with SOA, such as modularity and service-based interaction, but generally emphasize smaller independently deployable services, lightweight communication, and decentralized deployment. Giglio et al. [22] evaluate the use of IEC 61499 for process control using microservices, Eclipse 4diac, and OPC UA in the context of Industry 4.0. Pontarolli et al. [34] present a microservice-oriented architecture for Industry 4.0 applications, focusing on the evolution of Industrial Automation Systems through modular services, communication infrastructure, and process-control experimentation. These works reinforce the continuing relevance of IEC 61499, SOA, and microservice-oriented approaches for industrial systems. At the same time, SOA and microservices should not be treated as equivalent: in this dissertation, SOA is used primarily as an architectural principle for abstraction, discovery, composition, and

loose coupling, while the implemented prototype remains focused on runtime management and binding of IEC 61499 pipelines.

Table 2.2: Comparison of representative SOA and microservice-based approaches in industrial automation.

Work	IEC 61499	SOA	Microservices	Edge-oriented
Dai et al. [10]	✓	✓	×	○
Jhunjhunwala, Atmojo, and Vyatkin [26]	✓	✓	×	○
Hästbacka et al. [24]	×	✓	×	✓
Breunig, Roedel, and Bauernhansl [6]	×	✓	×	✓
Zhang et al. [48]	×	✓	×	✓
Pontarolli et al. [34]	×	○	✓	×
Giglio et al. [22]	✓	○	✓	○

Legend: ✓ fully addressed; ○ partially addressed; × not addressed.

Table 2.2 shows that several works address service orientation in industrial automation, but they emphasize different aspects. Some works directly combine IEC 61499 and SOA, while others focus on edge-oriented service infrastructures or microservice-based industrial applications. This distinction is important because the present dissertation does not aim to replace existing runtime execution with a complete SOA or microservice infrastructure. Instead, it uses SOA principles to structure the orchestration of distributed IEC 61499 runtime capabilities.

The reviewed literature shows that SOA and IEC 61499 are complementary. IEC 61499 provides a distributed FB model, event/data interfaces, and management operations, while SOA provides principles for describing, discovering, and composing functionality. Existing work has demonstrated formal mappings, design guidelines, plug-and-play service components, smart sensor services, and microservice-based process-control examples [9–13, 22, 26, 34]. However, these works do not fully address the problem defined in Section 1.3: maintaining a persistent and runtime-accessible representation of DINASORE instances, deployed pipelines, registered services, and cross-pipeline bindings through an external orchestration layer.

SOA therefore contributes the conceptual mechanisms required to treat deployed FB instances as reusable capabilities. Service information can be stored and discovered through a registry, and bindings can be used to compose functionality across independently deployed pipelines. In practice, however, these principles require supporting infrastructure: service records, runtime state, communication endpoints, and binding information must remain available beyond the local execution process. Middleware and DT platforms are relevant in this context because they provide the persistence and integration mechanisms needed to operationalize service-oriented orchestration across heterogeneous components.

2.3 Middleware and DT Platforms

Middleware is commonly used to reduce the complexity of integrating heterogeneous devices, protocols, software components, and applications in distributed systems. In IoT and industrial IoT contexts, middleware can provide an intermediate layer between devices and applications, abstracting communication details and offering mechanisms for device management, data exchange, service discovery, service composition, and interoperability [25, 28]. This role is important in edge-centric industrial systems because runtime instances, dashboards, service registries, messaging infrastructures, and data stores may need to interact without being tightly coupled to a single vendor-specific platform.

IoT middleware is often discussed in terms of requirements such as scalability, heterogeneity management, device discovery, resource management, data management, code management, security, and protocol support [25, 28]. Held, Schauz, and Domaschka [25] emphasize that selecting an IoT middleware is difficult because no single platform satisfies all requirements for every use case. This is relevant to industrial automation because the middleware layer must be selected according to the type of integration required. In this dissertation, middleware is not used as a generic cloud integration layer, but as a way to support persistent representation, message exchange, and orchestration around IEC 61499 runtime instances.

Service-oriented IoT middleware provides an additional perspective by combining middleware functions with service-based abstractions. Patel and Modi [32] describe middleware as a key part of IoT architectures, providing an interface between heterogeneous devices and IoT applications while supporting properties such as scalability, reliability, availability, security, and privacy. These properties are relevant to the orchestration problem considered here, since DINASORE runtimes, the orchestration interface, and the persistent representation of runtime state require a stable interaction layer. However, the objective is not to adopt a complete IoT middleware stack, but to use middleware concepts selectively to support service-oriented IEC 61499 orchestration.

Message-oriented middleware is particularly relevant when distributed components interact by exchanging messages rather than by direct function calls. In industrial automation, this model supports loose coupling because software components do not need to maintain direct point-to-point dependencies. Ashiwal, Majumder, and Zoitl [1] evaluate middleware technologies for implementing the PLC-Service Bus in IEC 61499 and identify requirements such as platform independence, client library support, Quality of Service, open-source availability, and community activity. Their work is relevant because it explicitly connects middleware selection with IEC 61499 and service-bus-style interaction between control software components.

The PLC-Service Bus line of work is important because it addresses a problem similar to the motivation of this dissertation: reducing tight coupling between automation software components. The PLC-Service Bus decomposes control applications into modular software components that interact through a message bus, inspired by Enterprise Service Bus (ESB) concepts [1, 39]. IEC 61499 is considered suitable for this architecture because of its event-based execution, absence of global variables, and support for communication patterns such as event messages

and publish-subscribe interaction [1]. This supports the view that middleware can complement IEC 61499 by providing interaction mechanisms that are not fully covered by the standard alone.

Ashiwal et al. [2] extend this line of work by integrating Apache Kafka as a middleware layer into Eclipse 4diac for the PLC-Service Bus architecture. Kafka is used as a network layer to support message exchange and to take advantage of existing monitoring and analysis tools. This work demonstrates that mainstream middleware can be connected to IEC 61499 environments through suitable integration mechanisms. However, its focus is on message exchange between software components in a PLC-Service Bus, whereas this dissertation uses middleware mainly to maintain runtime state, support service registration, and orchestrate bindings between independently deployed DINASORE pipelines.

OPC UA has also been used as a middleware-related mechanism for configuring and integrating IEC 61499-based architectures. Prenzel, Zoitl, and Provost [36] propose an OPC UA-based configuration approach for the IEC 61499-based PLC-Service Bus, using OPC UA services to automate runtime configuration. This is relevant because it shows that middleware and information-model technologies can be used not only for communication, but also for configuration and management operations. In this dissertation, a similar motivation exists: the orchestration layer should reduce manual effort and provide a persistent view of runtime elements. The approach considered here is centered on DT representation and service-oriented bindings rather than OPC UA-based configuration.

DTs provide a complementary abstraction for representing physical or logical assets in a persistent digital form. In industrial and IoT systems, a DT can store the latest known state of an asset, expose it through Application Programming Interfaces (APIs), and allow other applications to interact with that representation without directly depending on the asset itself [4, 8, 17]. This concept is relevant to the dissertation because IEC 61499 runtimes and deployed pipelines are dynamic entities: their state, available services, bindings, and configuration relationships need to be represented beyond the local execution process. A DT platform can therefore act as a persistent representation layer for runtime instances and their associated service information.

Eclipse Ditto is an open-source framework for implementing the DT software pattern in IoT projects. It represents physical or logical assets as *things*, each with a persisted state, features, attributes, and access-control policies [4, 17]. Clients can interact with these representations through APIs and supported connectivity mechanisms such as Hypertext Transfer Protocol (HTTP), WebSockets, Message Queuing Telemetry Transport (MQTT), Advanced Message Queuing Protocol (AMQP), and Kafka [4, 17]. Ditto is relevant not because it defines the architecture, but because it provides an implementation platform for the persistent middleware layer. The architectural idea remains independent from Ditto: a different platform could be used if it supported equivalent persistent representation, API access, and event-based updates.

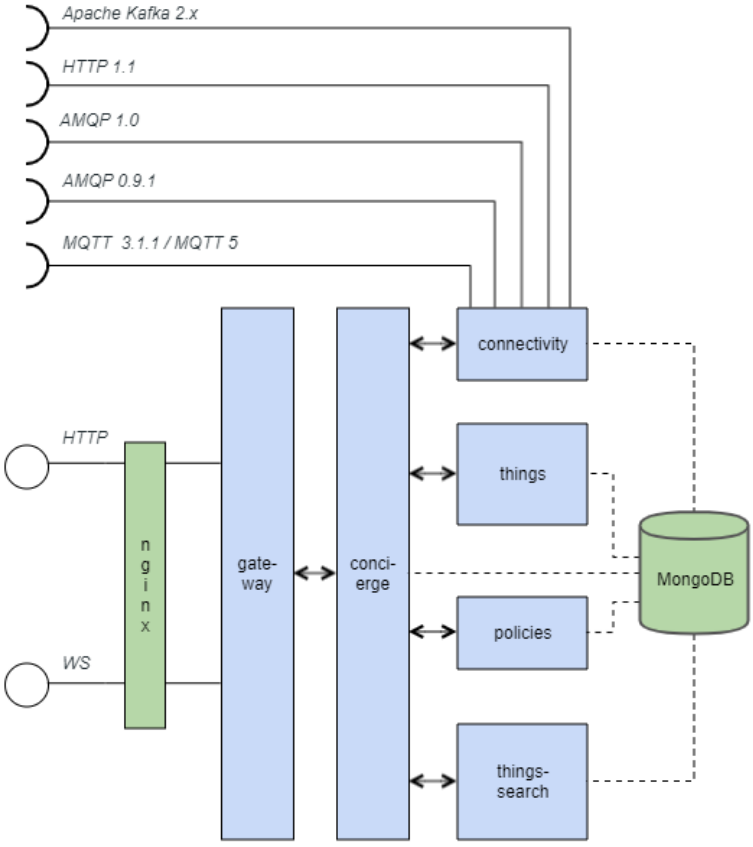


Figure 2.4: Eclipse Ditto architecture overview [17].

Figure 2.4 illustrates Eclipse Ditto as a middleware platform composed of several services for managing digital representations, policies, connectivity, and API access. In the context of this dissertation, the figure should be interpreted as an implementation background rather than as the proposed architecture itself. The relevant idea is that a persistent DT platform can maintain runtime-related state and expose it through APIs, while the actual IEC 61499 execution remains outside the platform.

The relation between service-oriented middleware and DTs is also visible in Arrowhead-related work. Eclipse Arrowhead is a service-oriented framework for industrial automation that organizes systems into local automation clouds and provides core services such as service registry, authorization, and orchestration [4, 14]. Aziz et al. [4] integrate Eclipse Ditto with Eclipse Arrowhead through a DT as a Proxy approach, using the DT to improve service availability, energy efficiency, and security for industrial cyber-physical systems. This work is useful for the present dissertation because it demonstrates that DTs can complement service-oriented industrial middleware by acting as persistent proxies or representations for distributed assets. Nevertheless, the architecture considered here does not adopt the full Arrowhead framework; it focuses on a smaller orchestration layer for DINASORE runtimes, service registration, discovery, and cross-pipeline binding.

Open-source DT platforms have also been compared from an implementation perspective.

Costa, Jorge, and Melo [8] evaluate platforms such as Eclipse Ditto, OpenTwins, Mainflux, FA3ST Tool Suite, and FIWARE, using criteria such as compatibility with DTs, implementation effort, documentation, scalability, protocol support, and licensing. Their case study selects Eclipse Ditto as the core of a simplified DT architecture. This supports the use of Ditto as a practical open-source platform for persistent asset representation, although the criteria in this dissertation are narrower: the platform must support persistent runtime state, API-driven interaction, and integration with the communication mechanisms used by the prototype.

Table 2.3: Comparison of representative middleware and DT approaches.

Work / Platform	IEC 61499	Message exchange	Service registry	Persistent state	Binding / orchestration
IoT middleware surveys [25, 28]	×	✓	○	○	○
Service-oriented IoT middleware [32]	×	✓	○	○	○
PLC-Service Bus middleware evaluation [1]	✓	✓	×	×	○
Kafka for PLC-Service Bus [2]	✓	✓	×	×	○
OPC UA configuration for PLC-Service Bus [36]	✓	○	×	○	○
Eclipse Arrowhead + Ditto [4]	×	✓	✓	✓	✓

Legend: ✓ fully addressed; ○ partially addressed; × not addressed.

Table 2.3 shows that middleware and DT approaches address complementary aspects of the problem. IoT middleware literature provides general requirements for communication, heterogeneity management, device discovery, and data management. PLC-Service Bus works connect middleware more directly with IEC 61499, but their focus is mainly message exchange and loose coupling between software components. DT-oriented work provides persistent state representation, while Arrowhead-like frameworks provide service registry and orchestration mechanisms. The proposed dissertation combines selected aspects of these directions: IEC 61499 runtime execution remains in DINASORE, while a persistent middleware-supported orchestration layer represents runtimes, pipelines, services, and bindings.

Taken together, the middleware and DT literature reinforces the separation between the main strands of existing work. IEC 61499 provides a distributed execution model, SOA provides principles for service abstraction and composition, and middleware platforms provide communication, persistence, and integration mechanisms. The combination required here remains insufficiently addressed: an edge-centric orchestration workflow for DINASORE-based IEC 61499 pipelines in which deployed FB instances can be registered as services, discovered, and bound across independently deployed pipelines through a persistent representation layer.

2.4 Comparative Analysis and Research Gap

The comparison between the reviewed strands of literature reveals a gap at the intersection of IEC 61499 runtime execution, service-oriented industrial automation, and middleware-supported DT representation. IEC 61499 provides modular and distributed FB-based execution, SOA supports abstraction, discovery, reuse, and composition, and middleware and DT platforms provide mechanisms for communication and persistent representation. However, these contributions are usually addressed separately.

Table 2.4: Comparative positioning of this dissertation against representative related work.

Approach / representative works	**IEC**	**Disc.**	**State**	**Bind.**	**Orch.**
IEC 61499 RTEs and engineering tools [23, 35, 49]	✓	×	×	×	○
DINASORE [33]	✓	×	×	×	○
SOA + IEC 61499 mappings [10–13]	✓	✓	○	○	○
Plug-and-play and smart sensor services [9, 26]	✓	✓	×	○	×
PLC-Service Bus and middleware integration [1, 2, 36]	✓	×	×	○	○
DT and industrial middleware platforms [4, 8, 17, 25, 28]	×	○	✓	○	○
This dissertation	✓	✓	✓	✓	✓

Legend: IEC = IEC 61499 support; Disc. = service discovery; State = persistent runtime state; Bind. = cross-pipeline binding; Orch. = unified orchestration workflow. ✓ fully addressed; ○ partially addressed; × not addressed.

Table 2.4 summarizes the capabilities covered by each group of related work. IEC 61499 runtimes and DINASORE support distributed execution, but they do not provide service discovery, persistent runtime state, or cross-pipeline service binding. SOA-related works show that IEC 61499 and service orientation are compatible, but they mainly focus on formal mappings, design guidelines, or specific service-enabled components. PLC-Service Bus and middleware approaches address loose coupling and message exchange, while DT and industrial middleware platforms provide persistence or service infrastructure. However, these works are not directly tailored to the orchestration of DINASORE-based IEC 61499 pipelines.

The research gap addressed in this dissertation is therefore the lack of an edge-centric, service-oriented orchestration architecture for IEC 61499 runtime instances in which deployed pipelines, FB instances, available services, and cross-pipeline bindings are represented persistently and exposed through a unified workflow. The proposed solution does not attempt to replace IEC 61499 execution semantics or to redefine DINASORE. Instead, it adds an orchestration layer, implemented through runtime-associated orchestrators, that allows runtime instances to be represented, services to be registered and discovered, and bindings to be created between independently deployed pipelines.

Chapter 3

Proposed Architecture

The research gap identified in Section 2.4, together with the problem and objectives established in Sections 1.3 and 1.4, motivates an edge-centric orchestration architecture for distributed IEC 61499 applications. The architecture is defined at the level of roles, information structures, interaction flows, and trade-offs, so that it remains independent from the specific technologies used in the prototype. Concrete choices such as DINASORE, Eclipse Ditto, MQTT, and the dashboard are treated as implementation decisions in Chapter 4, rather than as assumptions of the architecture itself.

3.1 Architectural Scope and Requirements

The architecture preserves the local execution of IEC 61499 applications inside each RTE. Time-critical FB execution, event propagation, and scheduling remain the responsibility of the runtime. The orchestration layer operates around those runtime instances, providing persistent representation, service registration, discovery, binding coordination, monitoring, and user interaction without redefining the IEC 61499 execution semantics.

This separation is central to the proposal. The runtime executes the control logic, while the orchestration layer coordinates the information and interactions required to discover, compose, and supervise distributed runtime capabilities. As a result, the architecture is not tied to a specific RTE, middleware platform, message broker, or dashboard implementation. These technologies instantiate the architecture in the prototype, but they are not part of its conceptual definition.

Table 3.1: Architectural requirements derived from the research gap.

Requirement	**Architectural mechanism**	**Related goal**
Runtime visibility	Each runtime instance is represented by a runtime descriptor containing its identity, endpoint, lifecycle state, and available capabilities.	Runtime representation and supervision
Persistent pipeline state	Deployed pipelines are represented independently from the runtime process, allowing their structure and state to remain inspectable outside the local RTE.	Persistent orchestration state
IEC 61499 integration	Pipeline changes are translated into the management operations already accepted by the target RTE, preserving local FB execution.	Runtime-compatible deployment
Service abstraction	Deployed FB instances are exposed through service descriptors containing type, interface, owning runtime, and availability information.	FB-level service representation
Registry separation	Runtime capabilities and deployed service instances are stored separately to avoid confusing available FB types with services that are already active.	Discovery correctness
Cross-pipeline binding	Directed binding descriptors coordinate source outputs, target inputs, and target triggering events across independently deployed pipelines.	Distributed composition
Unified orchestration workflow	Runtime discovery, pipeline deployment, service discovery, binding creation, monitoring, and user interaction are exposed through a common orchestration interface.	Engineering workflow integration
Edge-local operation	Orchestration components run within the edge infrastructure and do not require time-critical control execution to be moved to a remote cloud layer.	Edge-centric deployment

Table 3.1 links the requirements to the mechanisms used in the architecture. The main point is that the proposal is not only a communication mechanism between runtime instances. It also introduces persistent state, service descriptors, registry separation, and binding coordination, which together support an orchestration workflow around IEC 61499 runtimes.

3.2 Reference Orchestration Architecture

The reference architecture in Figure 3.1 organizes the proposed approach as a set of logical layers placed around distributed IEC 61499 RTEs, together with the implementation technologies adopted in each layer. At the bottom, each edge node hosts an IEC 61499 runtime that executes local FB networks. These FB instances can be exposed as services without changing the local execution responsibilities of the RTE. Above the runtime nodes, the orchestration layer is responsible for service exposure, binding management, and command translation between service-oriented operations and the management mechanisms supported by the runtime.

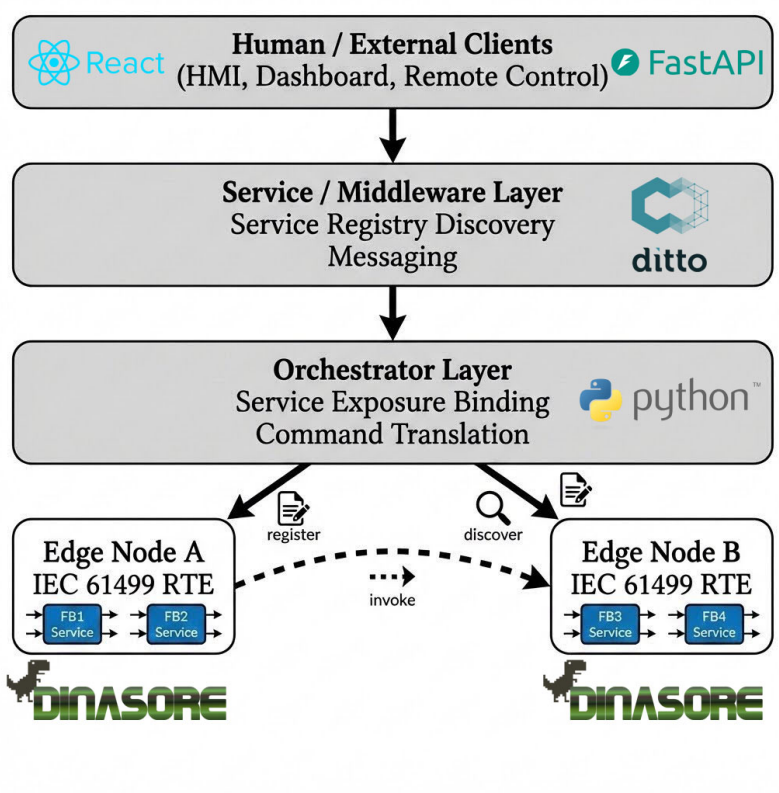


Figure 3.1: Reference architecture of the proposed service-oriented orchestration layer, illustrating both its logical organization and the implementation technologies adopted in the prototype.

The service and middleware layer provides the persistent coordination mechanisms required by the architecture, including service registry, discovery, and messaging support. Human users and external clients interact with this layer through interfaces such as dashboards, HMIs, or remote-control applications, instead of directly managing the communication details of each runtime instance. The figure also highlights the main service-oriented interactions supported by the architecture: deployed FB instances can be registered as services, discovered through the middleware-supported service layer, and connected through bindings or invocations across edge nodes.

This separation establishes a clear boundary between execution and orchestration. The IEC 61499 RTE remains responsible for executing FB networks and interpreting runtime management commands, while the orchestration and middleware layers maintain service descriptions, discovery information, and binding state. As a result, the architecture can be described independently of a specific RTE implementation, provided that the concrete runtime exposes equivalent mechanisms for deployment, monitoring, and communication with the orchestration layer.

The logical roles of the architecture are grouped into seven components.

IEC 61499 runtime instance. The runtime is responsible for executing the deployed FB network. It creates FB instances, propagates events and data, executes local application logic, and exposes the management interface required to create, update, read, start, or remove runtime

elements. The proposed architecture does not modify the runtime scheduling or execution semantics.

Runtime-associated orchestrator. The orchestrator is the integration boundary between one runtime instance and the orchestration layer. It translates persistent representation changes into IEC 61499-compatible management requests, reflects runtime responses back into the persistent state, registers deployed FB instances as services, and coordinates local binding behavior.

Persistent representation layer. This layer stores runtime descriptors, pipeline representations, registry entries, and binding descriptors. Its purpose is to keep orchestration state available beyond the lifetime of a dashboard session or a local runtime message exchange. The implementation may use a DT platform, a database-backed middleware, or another persistent information model.

Capability and service registries. The architecture distinguishes between the capabilities a runtime can instantiate and the service instances that are currently deployed. This separation prevents a locally available FB type from being treated as an already deployed service.

Binding coordination mechanism. Bindings represent directed associations between producer and consumer services. The binding information identifies the source runtime, source FB output, target runtime, target FB input, and target event to be triggered after the value update.

Orchestration interface. The user-facing interface presents runtime topology, pipeline editing, service discovery, binding creation, and monitoring. Its role is to interact with the orchestration state and services, not to implement the runtime-specific protocol of every RTE.

Communication interfaces. The architecture assumes a combination of request-response and asynchronous communication. Runtime commands may be carried through an RTE-specific interface such as Transmission Control Protocol (TCP), while orchestration events, state updates, and binding messages may use middleware or message-oriented communication.

3.3 Information Model

The information model distinguishes runtime descriptors, pipeline descriptors, runtime capabilities, deployed services, and bindings. These elements are separated because each one answers a different orchestration question: which runtimes exist, what is deployed, what can be instantiated, what is available as a service, and how services are connected.

Runtime descriptor. A runtime descriptor identifies an IEC 61499 runtime instance and stores the metadata required for orchestration, such as identifier, endpoint, lifecycle state, and capability-publication status.

Pipeline descriptor. A pipeline descriptor represents the deployed FB network associated with a runtime. It includes FB instances, their parameters, and the IEC 61499 connections between them. Changes in this descriptor are interpreted by the corresponding orchestrator and translated into runtime management commands.

Capability registry. The capability registry stores the FB types that each runtime can instantiate. It supports pipeline editing because the user interface can load the block library according to the selected runtime.

Service registry. The service registry stores descriptors for FB instances that have already been deployed and exposed as services. Service discovery queries are resolved against this registry, not against the capability registry.

Binding descriptor. A binding descriptor stores the producer-consumer relationship required for cross-pipeline composition. It identifies the source value, target value, target event, and owning runtime instances.

Table 3.2: Generic service descriptor used to expose a deployed FB instance.

Field	Description
Runtime or pipeline identifier	Identifies the runtime or deployed pipeline that owns the FB instance.
FB type	Describes the FB type and the logical capability exposed by the service.
Inputs	Lists the input variables and their IEC 61499 data types.
Outputs	Lists the output variables and their IEC 61499 data types.
Events	Describes the input and output events exposed through the FB interface.
Status	Indicates the availability or lifecycle state of the service instance.

Table 3.2 shows the minimum information required to make a deployed FB instance discoverable. The descriptor does not replace the FB implementation; it provides the orchestration metadata needed for discovery, selection, and later binding with another runtime pipeline.

3.4 Interaction Flows

The architecture is defined by five main interaction flows. These flows describe how information moves between runtime instances, orchestrators, persistent representations, registries, bindings, and the user interface.

Runtime registration and capability publication. When a runtime and its associated orchestrator start, the orchestrator publishes runtime metadata and inspects the locally available FB type definitions. These definitions update the capability registry, allowing the orchestration interface to display the runtime and load its block library.

Pipeline deployment and reconfiguration. The user creates or modifies a pipeline through the orchestration interface. The persistent pipeline descriptor is updated and delivered to the orchestrator associated with the selected runtime. The orchestrator converts FB and connection descriptions into IEC 61499 management commands and sends them to the runtime. Once deployment succeeds, the deployed FB instances can be registered as services.

Service discovery. A consumer requests services matching an FB type or interface. The orchestration layer queries the service registry and returns compatible deployed services, including their owning runtime, exposed variables, events, and status. The consumer can therefore select a service without knowing the complete runtime topology in advance.

Cross-pipeline binding. A binding is created after a producer and consumer have been selected. The binding descriptor identifies the source output, target input, and target event. The producer-side orchestrator publishes relevant value changes, while the consumer-side orchestrator converts accepted updates into the runtime requests needed to update the target FB and trigger the selected event.

Monitoring and user interaction. The user can select runtime values for monitoring and activate supported events manually. The associated orchestrator creates the required runtime requests, reflects monitored values into the persistent representation, and propagates updates to the orchestration interface.

Together, these flows make the architecture more than a registry. The persistent representation layer stores runtime and pipeline state, while the orchestrators perform the translation and coordination needed to keep that state aligned with actual IEC 61499 runtime execution.

3.5 Architectural Trade-offs

The architecture introduces orchestration capabilities that are not available in the baseline runtime workflow, but those capabilities also introduce costs and limitations.

- **Middleware dependency.** Running pipelines remain local to each runtime, but discovery, reconfiguration, binding creation, and dashboard operations depend on the availability of the persistent representation and message infrastructure.
- **Additional resource consumption.** Persistent state updates, registry operations, orchestration messages, and monitoring requests introduce CPU, memory, and network overhead.
- **Non-deterministic cross-pipeline communication.** Middleware-supported bindings are appropriate for flexible composition and coordination, but they do not provide hard real-time timing guarantees.
- **Fine service granularity.** Representing each deployed FB instance as a service creates a direct mapping to IEC 61499, but it can increase registry size and orchestration traffic as the number of FBs grows.

- **Monitoring overhead.** Periodic monitoring simplifies the prototype implementation, but it may generate runtime requests even when values remain unchanged.
- **Registry consistency.** Runtime disconnections, repeated FB names, deleted pipelines, and partial failures require stable identifiers and explicit lifecycle handling to avoid stale or ambiguous registry entries.

These trade-offs define the limits of the architectural claim. The architecture aims to improve orchestration, service discovery, persistent representation, and composition around IEC 61499 runtimes; it does not claim to improve the deterministic execution of local control logic or to replace runtime-specific scheduling mechanisms.

Chapter 4

Prototype Implementation

This chapter describes the prototype implementation of the architecture defined in Chapter 3. The implementation follows the architectural roles introduced in Section 3.2, the information model defined in Section 3.3, and the interaction flows described in Section 3.4. DINASORE implements the IEC 61499 runtime role, Eclipse Ditto and MongoDB implement the persistent representation layer, Eclipse Mosquitto provides MQTT communication, Python components implement runtime-associated orchestration and dashboard bridging, and the React-based dashboard provides the user-facing orchestration interface.

4.1 Implementation Overview

The prototype combines the architectural roles from Chapter 3 with concrete technologies. DINASORE executes IEC 61499 FB pipelines. Eclipse Ditto and MongoDB maintain the persistent DT and registry state. Eclipse Mosquitto provides the MQTT broker used for asynchronous synchronization and binding messages. One Python orchestrator process is associated with each DINASORE instance, and a web dashboard provides the orchestration interface.

Table 4.1: Technology mapping used in the prototype implementation.

Architectural role	**Prototype technology**	**Implementation role**
IEC 61499 runtime	DINASORE	Executes the deployed FB pipelines and interprets the XML management commands sent through its runtime interface.
Persistent representation layer	Eclipse Ditto and MongoDB	Store pipeline Things, runtime metadata, registry entries, monitored values, and binding representations.
Message infrastructure	Eclipse Mosquitto	Carries Ditto events, orchestrator messages, monitored values, and binding updates through MQTT.
Runtime-associated orchestrator	Python orchestrator process	Translates DT changes into IEC 61499 XML requests, processes runtime responses, registers services, and coordinates bindings.
Orchestration interface	React dashboard and FastAPI bridge	Provides pipeline editing, topology visualization, service discovery, binding creation, and monitoring through HTTP and WebSocket communication.

Table 4.1 shows that the prototype is an implementation of the architectural roles rather than a redefinition of the architecture itself. The design remains centered on IEC 61499 runtime execution and service-oriented orchestration, while the selected technologies provide a practical way to evaluate the proposed workflow. Eclipse Ditto was selected as the Digital Twin platform because it is an open-source solution that provides persistent asset representation and integrates natively with MongoDB, which was therefore adopted as the persistence layer. Python was chosen for implementing the orchestration logic due to its direct compatibility with the DINASORE runtime and its extensive support for asynchronous communication and middleware integration. Finally, React was selected for implementing a web-based dashboard due to its mature ecosystem of graph visualization libraries and its ability to support real-time interaction with Eclipse Ditto through WebSocket communication, while FastAPI was adopted as its backend framework for implementing the corresponding Representational State Transfer (REST) APIs and WebSocket endpoints.

4.2 Middleware Deployment and Information Model

Eclipse Ditto and MongoDB are deployed through Docker containers. In addition to the standard deployment configuration, the prototype defines connection and policy files. The connection configuration routes MQTT messages between Eclipse Ditto and Eclipse Mosquitto, while the policy configuration grants the orchestrators and dashboard bridge the permissions required to read and modify the relevant Things.

The prototype uses MQTT version 5 as the main asynchronous communication path. HTTP is used for explicit retrieval and discovery queries, and WebSocket communication is used by the dashboard bridge for bidirectional updates and event delivery. This separation follows the interaction pattern of each operation: persistent state changes are stored in Ditto, event notifications are propagated asynchronously, and dashboard queries are resolved through request-response endpoints.

A DINASORE pipeline is serialized as a JavaScript Object Notation (JSON)-based Eclipse Ditto Thing. The Thing represents the runtime instance, while its Features represent the deployed FB instances and the IEC 61499 connections between them. Eclipse Ditto can create, retrieve, modify, merge, and delete this representation, thereby maintaining a persistent version of the orchestration state. These Eclipse Ditto operations are distinct from the IEC 61499 management commands executed by DINASORE. When a merge or modification event is received for a pipeline Thing, the orchestrator interprets the updated representation and generates the corresponding IEC 61499 XML commands.

Figure 4.1 summarizes the representation used by the prototype. At the runtime level, an IEC 61499 pipeline is composed of FB instances and their event/data connections. In Eclipse Ditto, this deployed structure is represented as a pipeline Thing containing runtime metadata, FB Features, and connection information. After deployment, service-relevant FB instances are exposed as descriptors in `soa:registry`. Bindings are represented separately as persistent Things that identify source services, target services, mapped variables, and the target event to be triggered after a received value is written.

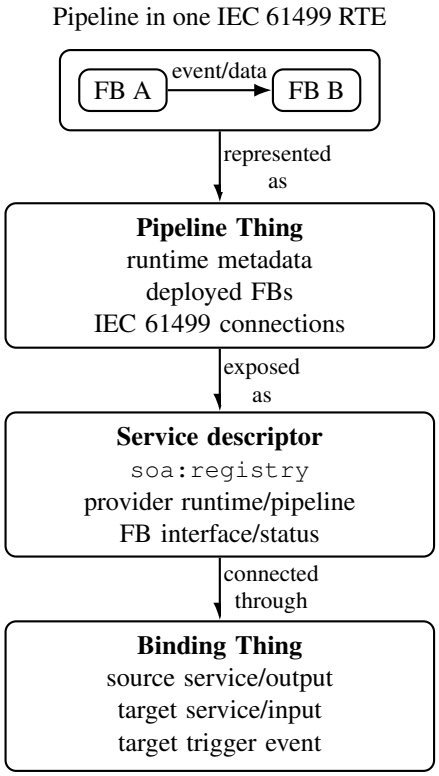


Figure 4.1: Mapping between IEC 61499 runtime structures and their Eclipse Ditto representations.

The complete JSON examples are placed in Appendix A to avoid interrupting the implementation narrative. Listing A.1 shows a complete pipeline Thing, including runtime attributes, FB Features, variables, events, and internal IEC 61499 connections. Listing A.2 shows the service-oriented representation used for a deployed FB instance inside `soa:registry`. Listing A.3 shows the binding configuration used to connect outputs from source services to inputs of a target service and to define the event that completes the target-side recomputation.

The implementation uses two registry Things. The `fb:registry` Thing contains the FB types available in each DINASORE installation, while `soa:registry` contains the FB instances that have been deployed and exposed as services. This separation implements the distinction introduced in Section 3.3: a local FB type is a runtime capability, whereas a deployed FB instance is a service candidate. Binding Things complement these registries by storing persistent cross-pipeline associations that can be interpreted by the participating orchestrators.

4.3 Runtime Integration and Orchestrator Implementation

The orchestrator implements the runtime-associated role defined in Section 3.2 and supports the deployment, monitoring, and binding flows described in Section 3.4. It connects the local DINASORE instance to Eclipse Ditto and maintains the DINASORE TCP socket, the MQTT client, request and response queues, and the service and binding state associated with the runtime. Its

core role is to translate between two representations: the persistent DT representation stored in Eclipse Ditto and the IEC 61499 XML commands interpreted by DINASORE.

Figure 4.2 shows how this translation is used during pipeline deployment and service registration. The workflow starts with the creation of an initially empty Thing in Eclipse Ditto to represent the DINASORE runtime instance. The dashboard then uses this representation to configure and deploy a pipeline. Once Eclipse Ditto emits a `modified` event, the orchestrator receives the updated configuration through MQTT, translates the deployed Features into DINASORE requests, and sends the corresponding XML commands through the TCP management interface. The same deployment information is also used to create service objects and publish their descriptors to `soa:registry`.

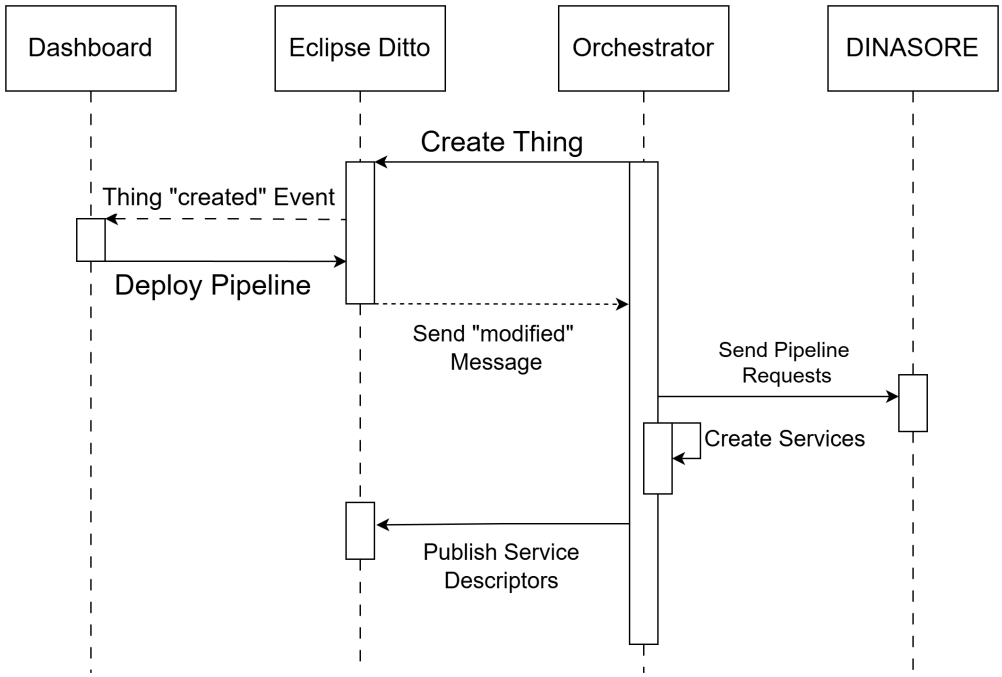


Figure 4.2: Sequence diagram of the deployment and service-registration workflow implemented by the prototype.

The sequence in Figure 4.2 highlights that deployment and service registration are handled as part of the same orchestration workflow. Eclipse Ditto stores the persistent representation of the pipeline, while DINASORE remains responsible for executing the resulting IEC 61499 FB network. This separation allows the prototype to add dashboard-supported deployment, persistent runtime representation, and service registration without modifying the execution responsibilities of the DINASORE runtime.

The `Request` and `Response` classes do not introduce new runtime operations. They provide structured storage and conversion for the IEC 61499 management messages already supported by DINASORE. A `Request` stores the command type and its parameters before serializing them into XML, while a `Response` stores the acknowledgements and monitored runtime values returned by DINASORE.

The prototype uses IEC 61499 commands such as `CREATE`, `WRITE`, `READ`, `DELETE`, and `START`. Configuration commands normally produce an `OK` acknowledgement, whereas monitoring commands return runtime values. The orchestrator uses these responses to advance deployment, update monitoring queues, and publish state changes back to Eclipse Ditto.

Runtime communication is divided into three dedicated processing threads, each implemented through a thread target function inside the orchestrator.

`self._listener_thread` Receives responses from DINASORE, processes acknowledgements, and places monitoring responses in `self.watch_queue` for publication.

`self._request_thread` Consumes `Request` objects, converts them to XML, and sends them through the DINASORE TCP connection. When monitoring is active, this thread also generates periodic `READ` requests.

`self._runtime_request_thread` Consumes updates received from bindings and creates the `WRITE` and event-trigger requests required to update the target FB.

The queues decouple message arrival from runtime processing. The thread-based structure allows deployment requests, monitored values, and binding updates to be processed concurrently without requiring a single blocking execution path inside the orchestrator.

Table 4.2: Main orchestrator methods used in the prototype.

Method	Purpose
`build_requests(self, data)`	Converts a complete pipeline representation into ordered FB and connection creation requests and appends the final `START` request.
`handle_merged(self, command)`	Processes a merged DT event, determines whether the pipeline already exists, creates service descriptors, and invokes the request-building procedure.
`append_bind(self, name, isOwner, bind=None)`	Creates the local binding state for the producer or consumer side and configures the monitored value associated with the binding.
`apply_bind_logic(self, resp)`	Checks whether a monitored source value changed and, when required, creates the update consumed by the runtime-request thread.
`get_local_fbt_names(self)`	Identifies the FB types available in the local DINASORE installation and prepares them for comparison with `fb:registry`.
`build_fb_registry_payload(self, fb_names)`	Builds the Feature payload used to create or update the FB-type entries associated with the DINASORE instance.

Table 4.2 summarizes the methods that implement the runtime-side orchestration workflow. They cover the translation from persistent pipeline state to runtime requests, the creation of service descriptors, and the local state required for cross-pipeline bindings.

The prototype preserves DINASORE's original FB execution semantics and TCP management interface. The adaptations are limited to startup and integration: each runtime is associated with an orchestrator, publishes metadata and available FB types, and uses the middleware-based deployment path instead of requiring Eclipse 4diac to remain responsible for prototype pipeline deployment. The startup configuration also includes the host name or Internet Protocol (IP) address used to identify the runtime endpoint published by each DINASORE instance.

Table 4.3: Command-line flags added to the DINASORE startup configuration.

Flag	**Purpose**	**Behavior**
`-d`	Defines the DINASORE instance identifier in Eclipse Ditto.	The identifier associates the runtime with its pipeline Thing, registry entries, service descriptors, and orchestrator.
`-host`	Defines the host or IP address associated with the DINASORE instance.	The address allows the Raspberry Pi deployments to publish the correct network endpoint for runtime communication and instance metadata.
`-sync`	Enables synchronization of the locally available FB types with `fb:registry`.	At startup, local FB definitions are compared with the entries stored for that DINASORE instance. New FB types are written to the persistent registry; when no changes are detected, startup continues without updating the database.

Table 4.3 shows that the DINASORE changes support orchestration rather than execution. The flags provide stable runtime identification, network configuration, and capability synchronization, while the scheduling and execution of the FB network remain unchanged.

4.4 Service Registration, Discovery, and Binding Implementation

The service registration, discovery, and binding mechanisms implement the service-oriented interaction flows introduced in Section 3.4. In the prototype, this behavior is implemented through three classes. `Service` stores the descriptor of a deployed FB instance, including its pipeline, FB type, variables, events, and status. `Bind` stores the source and target information required for a cross-pipeline association. `ServiceHandler` maintains the services and bindings associated with one orchestrator and coordinates their lifecycle.

After successful pipeline creation, each deployed FB is represented as a `Service` and published in `soa:registry`. Service discovery queries therefore target the deployed-service registry, while the dashboard block library targets `fb:registry`. This implementation avoids confusing FB types that a runtime can instantiate with FB instances that are already deployed and available for binding. The concrete JSON structure of a service descriptor is shown in Appendix A, Listing A.2.

Figure 4.3 summarizes the first part of the binding workflow. The dashboard queries `soa:registry` through Eclipse Ditto and receives the list of deployed services that can be used as binding endpoints. After the user selects the source and target services, the dashboard creates the binding

representation in Eclipse Ditto. The middleware then propagates the binding logic to the participating nodes: the consumer-side node subscribes to the binding-specific topic, while the provider-side node creates and stores the local `Bind` object required to monitor the selected output.

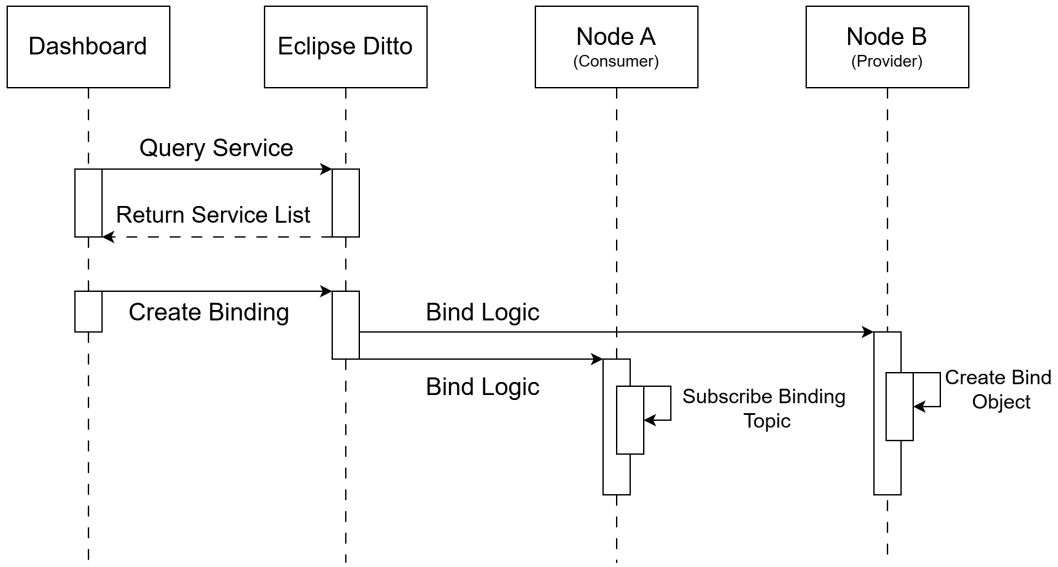


Figure 4.3: Sequence diagram of the service discovery and binding-creation workflow.

The binding configuration does not merge the participating pipelines into a single IEC 61499 application. Instead, it stores the variable mappings and the target trigger event as middleware state, as illustrated by the JSON example in Appendix A, Listing A.3. This allows independently deployed pipelines to remain separate while still sharing selected runtime values through the orchestration layer.

Figure 4.4 shows the second part of the binding workflow, in which the stored binding is applied at runtime. The provider-side node monitors the selected output through DINASORE WATCH/READ requests. When a relevant value changes, the provider-side node updates its local binding state and publishes the new value through MQTT. The consumer-side node receives the value, writes it to the configured target FB input, and triggers the event defined in the binding configuration so that the target FB recomputes its outputs using the updated values.

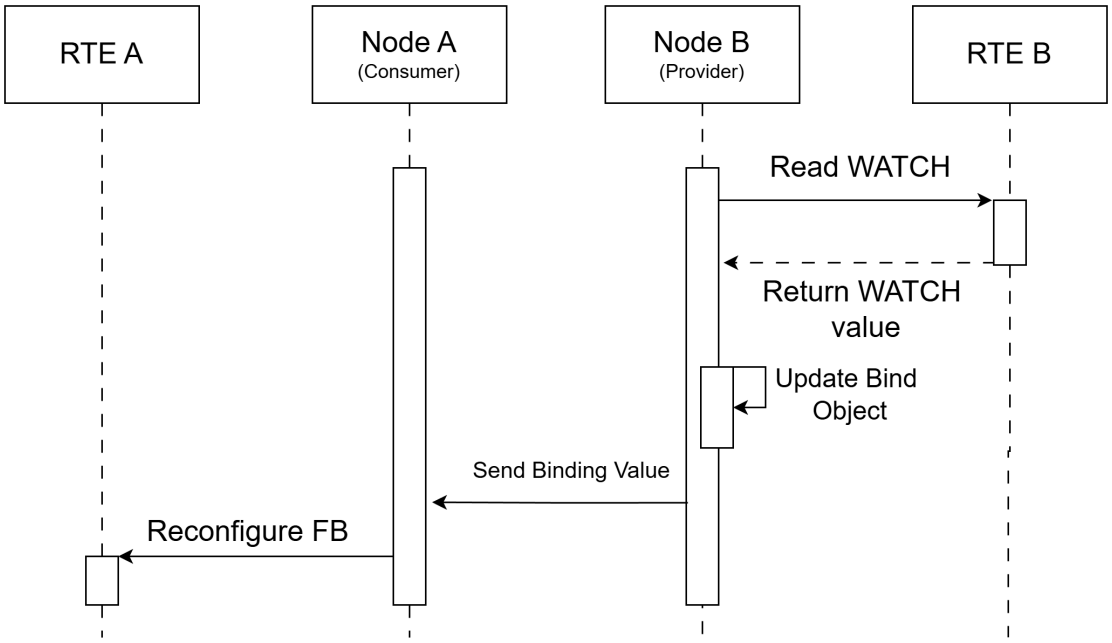


Figure 4.4: Sequence diagram of the runtime value-propagation workflow after a binding has been created.

The MQTT client subscribes to three logical topic groups. `reply_sub` receives Eclipse Ditto replies confirming that a command was accepted and processed. `event_sub` receives events associated with the runtime's pipeline Thing, including merged events for synchronization and deleted events for lifecycle handling. `bindings_sub` receives binding-creation messages and value updates exchanged between DINASORE instances.

The MQTT client also uses three dedicated processing threads, implemented through thread target functions. `_watch_consumer_loop()` is executed by the monitoring-publication thread and consumes monitored values before preparing middleware updates. `_mqtt_publish_loop()` is executed by the middleware-publication thread and publishes commands and state updates to Eclipse Ditto. `_bind_publish_loop()` is executed by the binding-publication thread and publishes producer values to the topics associated with active bindings. These threads separate runtime monitoring, middleware synchronization, and binding traffic.

4.5 Dashboard Implementation

The dashboard implements the orchestration interface defined in Section 3.2. It was implemented as a React-based interface. React Flow was used to support interactive node-edge editing of pipeline structures, while Reagraph was used to visualize graph-based relationships between runtime instances, services, and bindings [37, 47].

Communication between the dashboard and Eclipse Ditto is handled by `dittoBridge.py`, a FastAPI service that exposes HTTP endpoints for dashboard operations and a WebSocket endpoint

for asynchronous event delivery. The bridge maintains a WebSocket connection with Eclipse Ditto, subscribes to DT events, and broadcasts received messages to connected React clients. Dashboard requests that modify or retrieve DT state are forwarded through the Eclipse Ditto HTTP API.

Table 4.4: Endpoints exposed by `dittoBridge.py` for communication between the dashboard and Eclipse Ditto.

Endpoint	Method	Purpose
`/events`	WebSocket	Maintains the connection with the React dashboard and delivers connection notifications and DT events received through the Eclipse Ditto WebSocket interface.
`/registry`	GET	Retrieves the FB-type registry stored in `fb:registry`, allowing the dashboard to load the FB types available to the DINASORE instances.
`/status`	POST	Updates the `status` attribute of a DT, including lifecycle values such as `DEPLOYED` and `RUNNING`.
`/merge`	POST	Sends the updated Feature representation of a pipeline to Eclipse Ditto, synchronizing dashboard changes with the corresponding DT.
`/send-message`	POST	Sends a message to the inbox of a selected DT, for example to request runtime operations such as enabling the monitoring of selected FB variables.
`/query-service`	GET	Reads `soa:registry` and filters service descriptors according to the requested FB type, returning compatible deployed services to the dashboard.
`/create-thing`	POST	Creates a new Eclipse Ditto Thing from the supplied representation. In the dashboard workflow, this endpoint is used to create the DT that represents a binding.

Table 4.4 shows that the dashboard bridge does not execute IEC 61499 management commands or control FB scheduling. It provides the API boundary through which the dashboard reads and updates the persistent representations maintained by Eclipse Ditto.

The dashboard divides its main graph-based responsibilities between two views. `PipelineEditor` uses React Flow to represent FB instances as nodes and IEC 61499 connections as edges. It supports block insertion, parameter editing, pipeline deployment, and monitoring operations. `SystemTopology` uses Reagraph to represent the DINASORE instances discovered through Eclipse Ditto and the bindings established between them.

The remaining dashboard components support the surrounding orchestration workflow. `FunctionBlockProperties` displays and edits the selected FB parameters, configures pipeline connections, and presents monitored values. `BlockLibrary` loads the FB types available for the selected DINASORE instance from `fb:registry`. `InstanceMetadata` displays the instance identifier, management port, network address, and lifecycle state. `BindingCreation` configures a cross-pipeline binding by selecting the source output, target input, and target event. `QuerySe`

`rvicePanel` queries `soa:registry` through HTTP using a requested FB type and displays compatible services, owning runtimes, variables, and events.

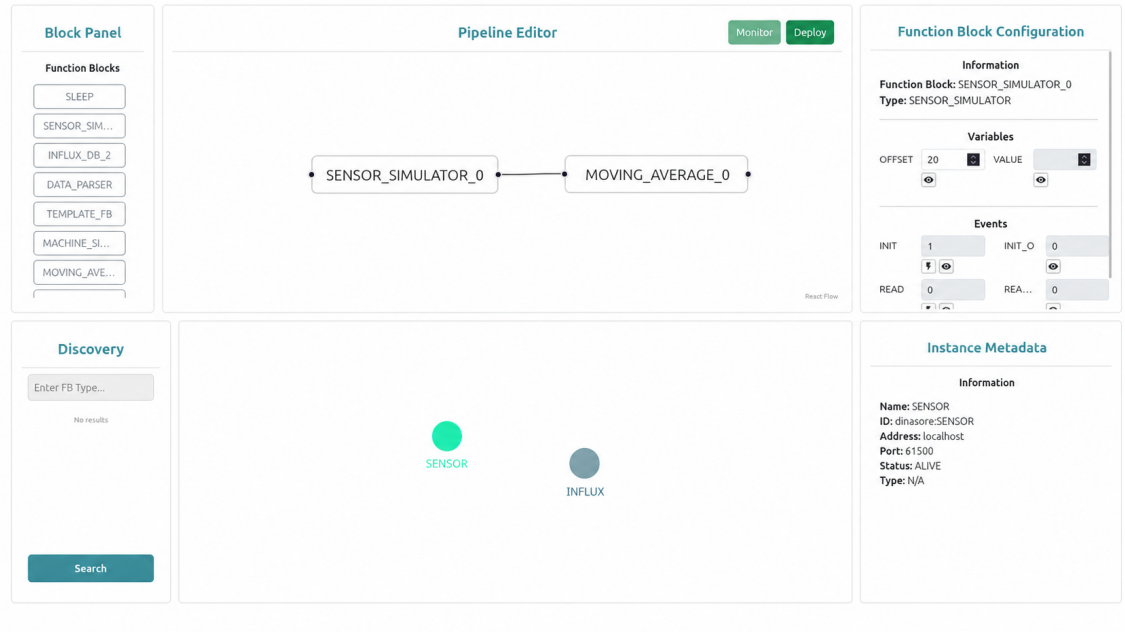


Figure 4.5: Composite dashboard view assembled from prototype screenshots, showing the configured sensor pipeline, FB properties, service discovery panel, runtime topology, and instance metadata.

Figure 4.5 illustrates how the dashboard brings the orchestration workflow into a single interface. The pipeline editor, metadata panel, service-discovery view, topology graph, and monitoring controls correspond to different parts of the architecture, but they operate over the same persistent middleware representation. This is why the dashboard can coordinate deployment, discovery, and binding without directly communicating with every DINASORE runtime.

4.6 Implementation Summary

The prototype implements the proposed architecture without replacing DINASORE's local execution engine. Eclipse Ditto stores the persistent pipeline, registry, and binding state; the orchestrator translates this state into IEC 61499 management operations; MQTT supports asynchronous synchronization and cross-pipeline bindings; and the dashboard provides a unified workflow for runtime discovery, pipeline deployment, monitoring, service discovery, and service composition.

The implementation therefore demonstrates the architectural separation introduced in Chapter 3. IEC 61499 execution remains local to each DINASORE instance, while the surrounding orchestration layer provides persistent representation, service registration, discovery, and binding support.

Chapter 5

Experimental Evaluation

This chapter evaluates the prototype implementation described in Chapter 4 in relation to the architecture defined in Chapter 3 and the objectives and research questions stated in Sections 1.4 and 1.4.1. The evaluation focuses on three complementary perspectives: runtime overhead, engineering workflow, and scalability. It is not intended to prove that the architecture improves industrial interoperability in a general sense. Instead, it examines whether the implemented prototype can support the representative scenario used in this work, which practical benefits it introduces over the traditional DINASORE workflow, and which computational and communication costs accompany those benefits.

5.1 Evaluation Methodology

5.1.1 Evaluation Scope and Scenarios

The evaluation was organized as an exploratory comparative study based on a representative distributed IEC 61499 application scenario. The experiments were designed to cover the main concerns raised by the architecture: the cost of adding the orchestration layer, the effect of the new workflow on application construction and configuration, and the behavior of the prototype when the number of runtime instances and bindings increases. Resource measurements focus on Central Processing Unit (CPU) and Random Access Memory (RAM) usage, while communication behavior is summarized through packet-success rate.

Table 5.1: Evaluation scenarios and metrics used in the experimental assessment.

Scenario	**Purpose**	**Metrics**
Single-pipeline runtime overhead	Compare the traditional and proposed workflows for the same distributed application scenario.	Scenario execution time, mean CPU usage, maximum RAM usage.
Comparative workflow and usability	Compare the engineering effort and perceived usability of the traditional DINASORE workflow and the proposed dashboard workflow.	Completion time, questionnaire responses, qualitative feedback.
Scalability Test 1: multiple instances on one Raspberry Pi	Assess how the prototype behaves when several DINASORE instances and associated orchestrators run on the same edge device.	Mean CPU usage, maximum RAM usage, packet-success rate.
Scalability Test 2: bound pipelines on two Raspberry Pis	Assess the cost of creating bindings between sensor-side and database-side pipelines distributed across two edge devices.	Mean CPU usage, maximum RAM usage, packet-success rate.

Table 5.1 separates the evaluation into scenarios instead of treating the prototype as a single monolithic test. This distinction is important because each scenario measures a different aspect of the architecture. Runtime overhead focuses on the additional cost of the orchestration infrastructure, the usability study focuses on the engineering workflow, and the scalability tests focus on how the edge devices behave as the number of instances and bindings increases.

5.1.2 Experimental Environment

All experiments were conducted in a local network composed of two Raspberry Pi 5 edge devices, one local machine, and an ASUS RT-N12 router connected through Ethernet cabling. The Raspberry Pi devices hosted the DINASORE runtime instances used by the distributed application, while the local machine hosted Eclipse Ditto, Eclipse Mosquitto, the dashboard, and the tools used to manage and observe the experiments. The setup description focuses on the hardware and operating-system information required to interpret the resource and scalability results.

Table 5.2: Experimental setup used in the evaluation.

Component	Role	Configuration
Two Raspberry Pi 5 Model B Rev. 1.1 devices	Edge devices hosting the sensor-side and database-side DINASORE runtime instances and their associated orchestrators.	ARM Cortex-A76, 4 cores, up to 2.4 GHz; 15 GiB RAM; Debian GNU/Linux 13 (trixie).
Local machine	Host for Eclipse Ditto, Eclipse Mosquitto, the dashboard, and experiment-control tools.	ASUS VivoBook X515EA/F515EA; Intel Core i7-1165G7 @ 2.80 GHz; 15 GiB RAM; Ubuntu 24.04.4 LTS.
Local network	Interconnection between the experimental devices.	ASUS RT-N12 Wireless N Router; Ethernet cables used for the experimental connections.

Table 5.2 summarizes the hardware and software elements needed to interpret the evaluation results. The two Raspberry Pi devices used the same model, processor class, memory capacity, and operating system, which avoids introducing a major hardware difference between the sensor-side and database-side runtime hosts. The local machine hosted the middleware and dashboard infrastructure, so the results should be interpreted as a controlled laboratory prototype deployment rather than as an industrial deployment benchmark.

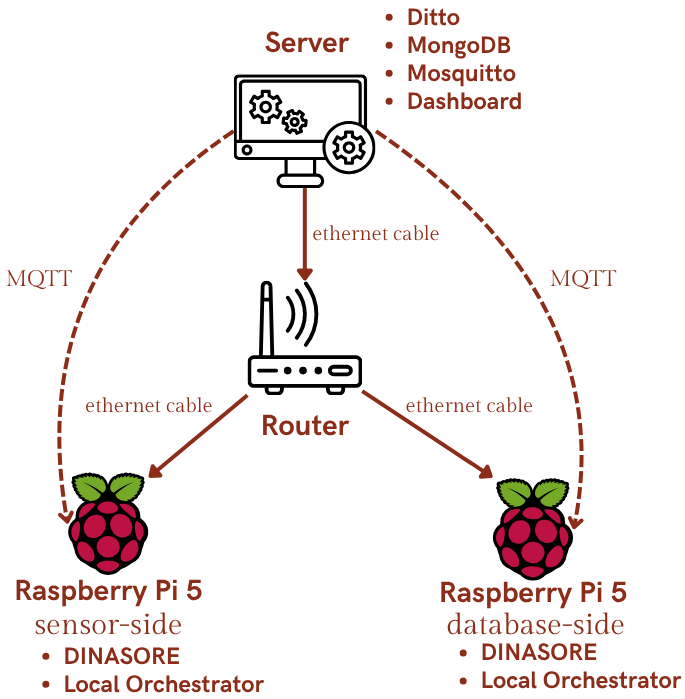


Figure 5.1: Experimental setup used to evaluate the proposed architecture, showing the deployment of the middleware infrastructure on the server and the distributed execution of DINASORE runtimes and local orchestrators on the edge devices. Solid arrows represent the physical Ethernet connections, while dashed arrows indicate MQTT communication.

Figure 5.1 complements the setup description by illustrating both the physical deployment of the proposed architecture and the communication infrastructure used during the experiments. The server hosts the shared middleware components, including Eclipse Ditto, MongoDB, Eclipse Mosquitto, and the dashboard, while each Raspberry Pi executes a DINASORE runtime together with its associated local orchestrator. The solid arrows represent the physical Ethernet connections established through the local router, whereas the dashed arrows indicate the MQTT communication flows exchanged between the server and the distributed edge nodes over the same local network.

Router-level configuration and network-level measurements such as latency, jitter, throughput, signal strength, and router load were not collected separately. The packet-success rate reported in the scalability experiments should therefore be interpreted as an application-level communication indicator for this setup, not as a complete network-performance characterization.

5.1.3 Application Scenario and Comparative Workflow

The evaluation used a distributed sensorization scenario adapted from DIGI2-FEUP [15]. On the sensor side, `SENSOR_SIMULATOR` generated values and `MOVING_AVERAGE` calculated the average over a configurable window. On the database side, `INFLUX_DB_2` stored the received sensor and moving-average values.

The same logical application was constructed through two workflows. In the traditional DINASORE workflow, participants used Eclipse 4diac to configure two runtime devices, create and map the FB network, and deploy it. Since the pipelines were hosted by different runtime instances, `MQTT_PUBLISHER_2` and `MQTT_SUBSCRIBER_2` were added explicitly to exchange values between them.

In the proposed workflow, participants used the dashboard to select the DINASORE instances, create and deploy the two pipelines, discover the database service, and define a binding between the source outputs and the target inputs and event. The binding replaced the explicit MQTT publisher and subscriber FBs in the application model. Both procedures used the same parameter values and produced the same data flow.

This scenario exercises the deployment, monitoring, service-discovery, and binding mechanisms defined at the architectural level in Section 3.4 and implemented in Sections 4.3 and 4.4.

Completion time was measured from the participant's start indication until the two pipelines had been deployed and their communication had been configured. After the timer was stopped, the participants were shown how WATCH operations, event triggering, and value updates were exposed by each interface. This monitoring walkthrough was not treated as a separate experimental stage and was not included in the completion-time measurements. Its purpose was to give participants enough context to answer the monitoring-related questionnaire item.

A condensed version of the participant protocol is provided in Appendix B. The appendix summarizes the tasks, parameters, stopping conditions, and information shown to participants, while omitting repeated button-level instructions and screenshots that are not necessary for reproducing the methodology. The original full guide is retained as supplementary project documentation.

5.1.4 Scalability Procedure

The scalability evaluation consisted of two tests. Both tests were automated through Python scripts that emulated the message exchange normally produced by the dashboard when interacting with Eclipse Ditto. Instead of requiring the user to manually create and orchestrate hundreds of pipelines through the graphical interface, the scripts issued the corresponding middleware-level operations used to create pipeline representations, configure monitoring, trigger execution, and create bindings. This made the construction of the scalability test environment repeatable and reduced the manual effort required to evaluate large numbers of runtime instances.

In Test 1, an increasing number of DINASORE instances was executed on a single Raspberry Pi. For each magnitude, the Python script created the instances, configured a WATCH operation, triggered each pipeline, enabled monitoring, and kept the pipelines active for 15 seconds. In Test 2, the same number of instances was created on each Raspberry Pi and one Eclipse Ditto binding was created for each sensor/database pair. After the bindings were established, the sensor pipelines were triggered and the configuration was kept active for 15 seconds.

Therefore, the scalability tests evaluate the orchestration backend, the middleware communication path, and the runtime-associated orchestrators under dashboard-equivalent commands, but they do not measure the rendering performance or manual interaction latency of the dashboard interface itself.

The tested magnitudes were 1, 5, 10, 20, 50, 100, 200, 300, 400, 500, and 600 instances. Higher magnitudes were initially planned, but they were not executed after the 600-instance configuration failed. Between magnitudes, the scripts waited for CPU and memory usage to return to a stable baseline before continuing. The 600-instance rows in Tables 5.5 and 5.6 are therefore marked as failed. CPU and RAM show the last values observed before failure, while packet-success values are omitted because the incomplete executions did not produce a consistent measurement window.

5.1.5 Metrics and Data Collection

The evaluation combined quantitative and qualitative metrics. Mean CPU usage was collected with `nmon`; mean usage was selected instead of peak usage to reduce the influence of short transient spikes. Maximum RAM usage was collected through operating-system commands and represents the highest observed memory occupation during each run. Packet-success rate was calculated from packet captures collected with `tshark`. The captures covered DINASORE TCP communication and MQTT traffic between the edge devices and the local machine.

Completion time measured the time required to construct, deploy, and connect the two pipelines in the comparative workflow study. Questionnaire responses captured participants' assessment of pipeline construction, monitoring, instance management, service discovery, binding creation, available functionality, and learning effort.

The usability study involved seven participants. Five participants (71.4%) had previous experience with Eclipse 4diac and/or DINASORE, while two participants (28.6%) had no previous

experience. Six participants were master's students and one was a doctoral student. The participant sample was therefore small and predominantly academic, so the usability results should be interpreted as exploratory evidence rather than as a general industrial-user evaluation.

Table 5.3: Questions used in the comparative usability questionnaire.

ID	Type	Evaluation criterion
Q1	Background	Previous experience with Eclipse 4diac for pipeline creation
Q2	Comparison	Pipeline construction process
Q3	Comparison	Platform user experience
Q4	Comparison	Monitoring operations, including WATCH creation and event triggering
Q5	Comparison	DINASORE instance management
Q6	Comparison	Available functionality
Q7	Comparison	Ease of connecting independent pipelines
Q8	Comparison	Support for constructing multiple pipelines
Q9	Comparison	Usefulness of bindings between pipelines
Q10	Comparison	Service and DINASORE instance discovery
Q11	Comparison	Perceived learning effort

The questionnaire in Table 5.3 distinguishes background information from comparative assessment. Q1 characterizes prior experience, while Q2–Q11 use a five-point comparative scale in which 1 represents a preference for the traditional Eclipse 4diac workflow and 5 represents a preference for the proposed dashboard workflow.

5.2 Results and Analysis

5.2.1 Runtime Overhead

The single-pipeline experiment compared the traditional and proposed workflows for the same distributed application scenario.

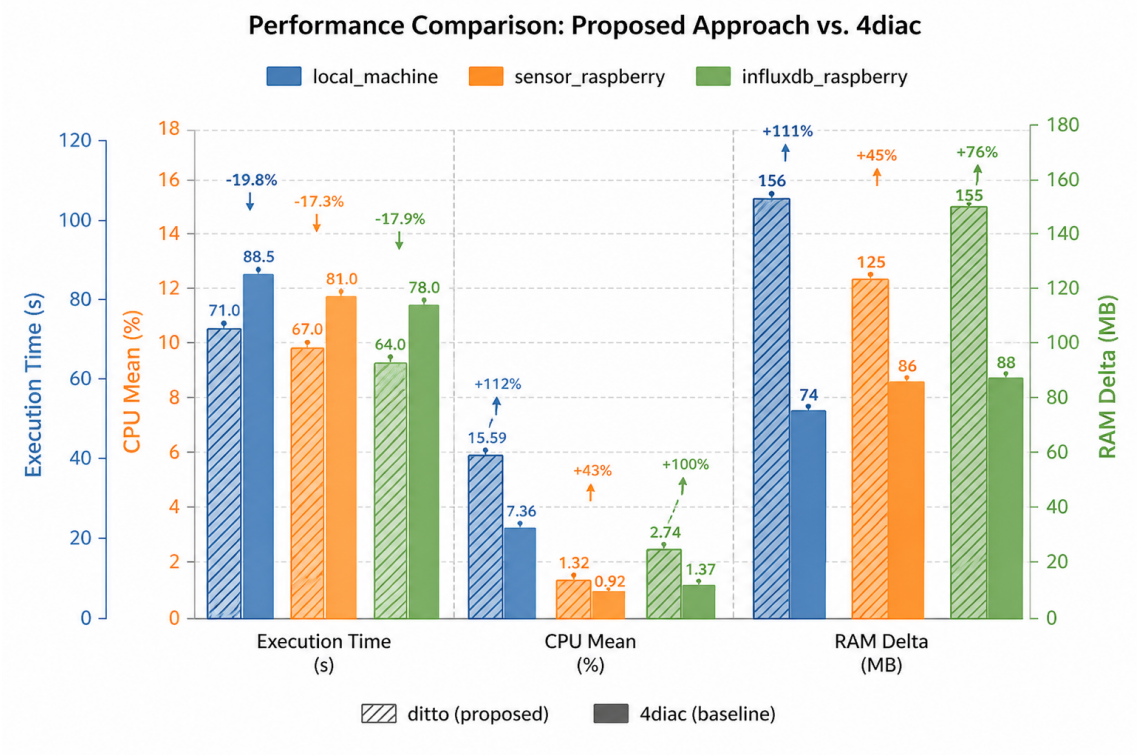


Figure 5.2: Execution time and resource overhead of the traditional and proposed workflows for the single-pipeline scenario.

Figure 5.2 shows that the proposed workflow completed the measured scenario faster on all three devices. Execution time decreased from 88.5 to 71.0 seconds on the local machine, from 81.0 to 67.0 seconds on the sensor Raspberry Pi, and from 78.0 to 64.0 seconds on the database Raspberry Pi. These reductions, between 17.3% and 19.8%, are consistent with a shorter workflow in which the application does not require explicit publisher and subscriber FBs. The experiment does not isolate the contribution of each removed or automated step, so the reduction should be interpreted as an end-to-end workflow result rather than as a runtime-execution speedup.

The shorter completion time was obtained at the cost of higher resource usage. Mean CPU usage increased from 7.36% to 15.59% on the local machine, from 0.92% to 1.32% on the sensor Raspberry Pi, and from 1.37% to 2.74% on the database Raspberry Pi. Additional memory usage was also higher in the proposed workflow: 156 MB instead of 74 MB on the local machine, 125 MB instead of 86 MB on the sensor device, and 155 MB instead of 88 MB on the database device. This overhead is consistent with the additional orchestrator logic, MQTT communication, DT synchronization, and monitoring support introduced by the architecture.

The runtime-overhead result therefore presents a trade-off rather than an overall performance improvement. The proposed workflow reduced the time required to configure and connect the evaluated application, but it consumed more CPU and memory, which is particularly relevant for resource-constrained edge devices.

5.2.2 Usability and Workflow

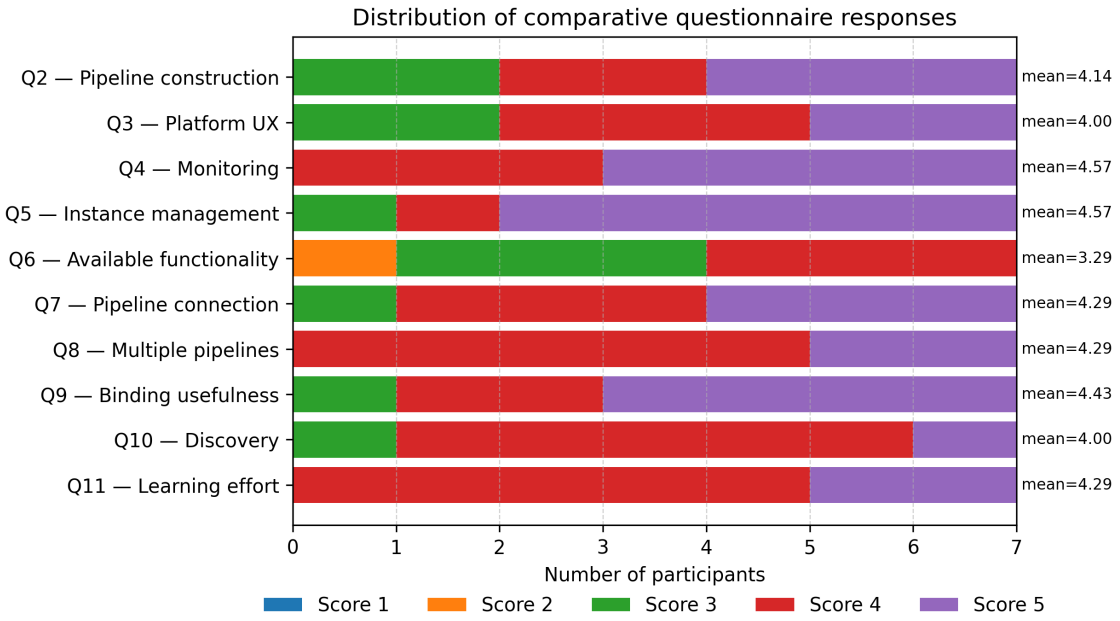


Figure 5.3: Distribution of comparative questionnaire responses from the seven participants.

Figure 5.3 shows the distribution of responses for the comparative questionnaire items. The strongest preference for the proposed workflow appears in monitoring and instance management, with Q4 and Q5 both reaching a mean score of 4.57. Binding usefulness also received a high score (Q9 = 4.43), while pipeline connection, support for multiple pipelines, and perceived learning effort each reached 4.29. Pipeline construction (Q2 = 4.14), general user experience (Q3 = 4.00), and discovery (Q10 = 4.00) were also evaluated positively.

Available functionality received the lowest comparative score (Q6 = 3.29). This result prevents the usability outcome from being interpreted as an unqualified preference for every aspect of the prototype. The distribution also shows that the participants did not evaluate all aspects uniformly, reinforcing that the proposed workflow was considered useful for orchestration, discovery, monitoring, and binding tasks, but that the interface and available functionality still require further refinement.

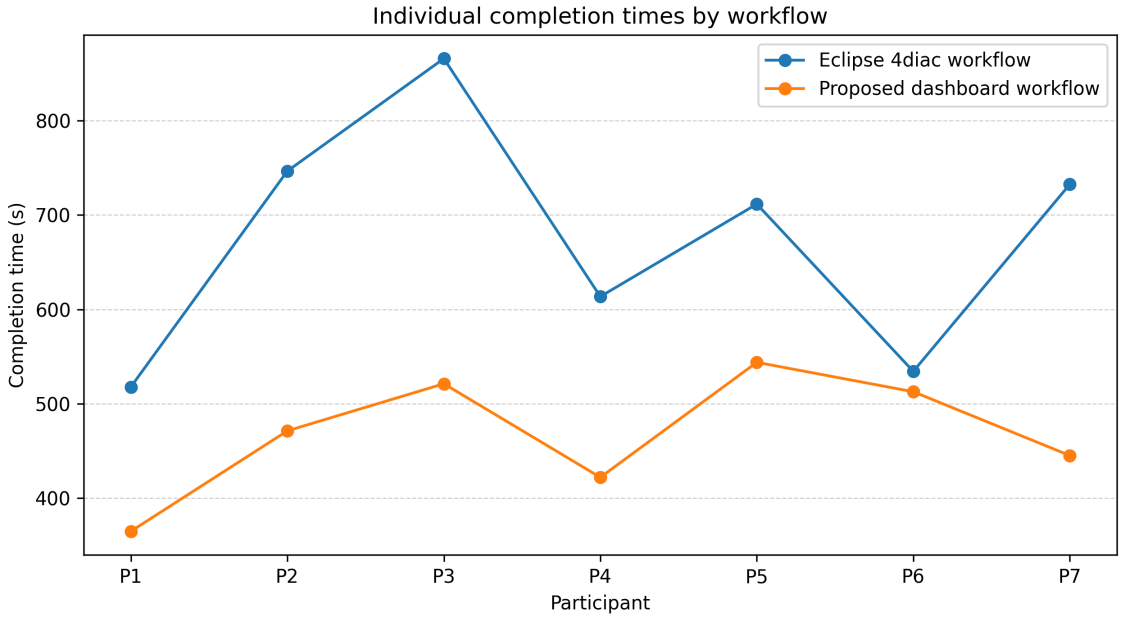


Figure 5.4: Individual completion times required by participants to construct and deploy the comparative application scenario.

Figure 5.4 shows the individual completion times measured for each participant in both workflows. The mean completion time decreased from 674.64 seconds in the Eclipse 4diac workflow to 468.52 seconds in the proposed dashboard workflow, corresponding to a reduction of 206.12 seconds, or 30.6%. Every participant completed the task faster with the proposed workflow, although the smallest individual difference was 21.62 seconds. This result supports the argument that service discovery, direct pipeline deployment, and middleware-managed bindings can reduce engineering effort for the evaluated task. However, the result is limited to the tested scenario and to a small participant sample, so it should not be generalized to all IEC 61499 engineering activities.

The open feedback provides context for the questionnaire scores. The most frequent suggestions concerned a more direct discovery interface, better visual feedback during binding deployment, WATCH creation from the pipeline editor, clearer variable names, and closer integration between pipeline editing and FB configuration.

Table 5.4: Qualitative feedback themes and the evaluation criteria to which they relate.

Feedback theme	**Related questions**
Dropdown-based or otherwise simplified service selection and improved search	Q10
Easier navigation to the runtime and pipeline that provide a discovered service	Q10
WATCH creation directly from the pipeline editor	Q4
Visual feedback while binding metadata is being deployed	Q4, Q9
Closer integration between pipeline editing and FB configuration	Q3, Q6
Improved visibility of variable names and parameters	Q3
More visual and intuitive interaction mechanisms	Q3
Further refinement and completion of the user interface	Q3, Q6
Additional functionality and shorter interaction workflows	Q6

The feedback summarized in Table 5.4 reinforces the quantitative results while also identifying limitations of the prototype interface. Participants valued discovery, binding, and monitoring support, but their comments indicate that the dashboard still requires better navigation, clearer deployment feedback, and more complete editing functionality before it could replace a mature engineering tool.

5.2.3 Scalability

Test 1: Multiple instances on one Raspberry Pi. The first scalability experiment concentrated the DINASORE instances and their associated orchestration processes on a single Raspberry Pi. The local columns in Table 5.5 refer to the machine hosting Eclipse Ditto and Eclipse Mosquitto, while the edge columns refer to the Raspberry Pi executing the DINASORE instances and orchestrators.

Table 5.5: Scalability Test 1: resource usage and packet-success rate with multiple DINASORE instances on one Raspberry Pi. CPU and RAM values for the failed 600-instance run are the last values observed before termination.

Instances	**Status**	**Ditto host**			**Raspberry Pi**		
		CPU (%)	**RAM (%)**	**Success (%)**	**CPU (%)**	**RAM (%)**	**Success (%)**
1	Completed	8.91	45.37	100.00	1.29	6.26	100.00
5	Completed	8.91	44.67	100.00	2.72	6.94	100.00
10	Completed	8.92	44.38	100.00	4.38	8.11	100.00
20	Completed	8.93	44.38	100.00	7.40	9.59	100.00
50	Completed	8.95	44.17	100.00	13.84	15.20	100.00
100	Completed	8.96	44.72	100.00	19.60	23.55	100.00
200	Completed	8.99	45.01	100.00	29.76	41.08	98.61
300	Completed	8.98	45.10	99.99	36.99	57.40	98.23
400	Completed	8.85	47.33	99.98	43.99	75.28	97.10
500	Completed	8.51	48.63	99.09	46.77	85.06	96.66
600	Failed	7.93	48.18	–	73.24	98.22	–

Table 5.5 shows that packet delivery remained effectively complete up to 100 instances and that Raspberry Pi resource usage increased gradually. From 200 instances onwards, the edge device showed a clearer loss of headroom: mean CPU increased from 29.76% at 200 instances to

46.77% at 500, while maximum RAM increased from 41.08% to 85.06%. Packet success on the Raspberry Pi also decreased from 98.61% at 200 instances to 96.66% at 500.

The local machine followed a different pattern. CPU remained close to 9% and RAM remained between approximately 44% and 49% through the completed runs. Packet success remained above 99%, including at 500 instances. These values indicate that the first practical bottleneck in this setup was the Raspberry Pi rather than the machine hosting Eclipse Ditto.

The 600-instance configuration did not complete consistently. The Raspberry Pi reached 98.22% RAM and 73.24% mean CPU before failure. The CPU and RAM values are useful as evidence of resource exhaustion, but the associated packet measurements and the lower local-host CPU value must not be interpreted as successful steady-state behavior. The experiment therefore does not establish a general architectural limit of 500–600 instances. It establishes that the tested Raspberry Pi configuration exhausted its available memory around that load.

Test 2: Bound pipelines on two Raspberry Pis. The second scalability experiment distributed the application across two edge devices. Each magnitude in Table 5.6 represents that number of sensor-side instances, the same number of database-side instances, and the corresponding number of bindings.

Table 5.6: Scalability Test 2: resource usage and packet-success rate for bound pipelines distributed across two Raspberry Pis. CPU and RAM values for the failed 600-instance run are the last values observed before termination.

Instances	Status	Ditto host			Sensor Raspberry Pi			Database Raspberry Pi		
		CPU (%)	RAM (%)	Success (%)	CPU (%)	RAM (%)	Success (%)	CPU (%)	RAM (%)	Success (%)
1	Completed	12.08	34.16	100.00	1.49	4.03	100.00	3.12	5.01	100.00
5	Completed	12.14	30.60	99.99	2.40	4.64	100.00	2.30	5.44	100.00
10	Completed	12.17	30.64	100.00	4.01	5.95	100.00	3.41	6.13	100.00
20	Completed	12.22	31.23	100.00	6.22	7.41	100.00	5.49	7.76	100.00
50	Completed	12.38	32.02	100.00	11.54	12.68	100.00	9.27	11.96	100.00
100	Completed	12.53	33.97	99.59	16.88	21.53	96.09	14.88	19.56	97.10
200	Completed	12.85	35.53	99.26	26.49	39.03	93.77	48.10	34.00	94.28
300	Completed	13.14	37.96	99.31	38.30	57.33	88.79	57.94	49.68	93.57
400	Completed	12.94	39.62	99.44	42.87	74.84	91.82	44.60	62.44	94.97
500	Completed	14.32	42.09	99.47	53.62	92.54	92.53	63.04	79.12	94.89
600	Failed	7.18	37.35	–	44.69	98.58	–	40.16	88.54	–

Table 5.6 shows that the binding scenario produced a higher load on both edge devices than Test 1 because each sensor pipeline exchanged values with a corresponding database pipeline. At 500 instances, the sensor Raspberry Pi reached 53.62% mean CPU and 92.54% RAM, while the database Raspberry Pi reached 63.04% mean CPU and 79.12% RAM. Packet success at the edge was 92.53% and 94.89%, respectively. The decline in packet success as load increased shows that the communication path was already under pressure before the final failure.

The machine hosting Eclipse Ditto remained comparatively stable through the completed runs. At 500 bindings, it used 14.32% mean CPU and 42.09% RAM, and its packet-success rate remained at 99.47%. This does not prove that Eclipse Ditto would scale without limit, but it shows that the middleware host was not the first saturated component in the tested configuration.

The 600-instance configuration failed when the sensor Raspberry Pi reached 98.58% RAM; the database Raspberry Pi had reached 88.54%. The lower CPU values recorded during this run are a consequence of premature termination and must not be interpreted as improved efficiency. As in Test 1, the row records the resource level reached before failure, while packet-success values are omitted. The result identifies the memory capacity of the edge hardware and the cost of running many orchestrated pipelines as the main constraints of this experiment.

Taken together, the experimental results indicate that the proposed workflow is feasible in the evaluated scenario and can reduce the manual effort required to construct and connect distributed pipelines. They also show that these benefits are obtained with additional resource consumption and that scalability is limited by the memory available on the edge devices. The results support the usefulness of the orchestration workflow for the tested setup, but they should be interpreted together with the limitations discussed later in the dissertation, especially the small participant sample, the academic profile of the participants, the absence of industrial users, and the use of a single validation scenario.

Chapter 6

Discussion

This chapter interprets the experimental results reported in Section 5.2 in relation to the objectives and research questions defined in Sections 1.4 and 1.4.1. The goal is to discuss what the results indicate about the use of service-oriented orchestration around IEC 61499 runtime instances, using DINASORE as the concrete RTE case study. The chapter also clarifies when the proposed architecture is appropriate, when it is less suitable, which benefits and costs were observed, and which limitations must be considered before drawing broader conclusions.

6.1 Interpretation of the Results

The main finding of this dissertation is that service-oriented orchestration can be introduced around DINASORE without replacing its local IEC 61499 execution model. The architecture defined in Sections 3.1, 3.2, 3.3, and 3.4 separates local control execution from orchestration concerns. DINASORE remains responsible for executing the deployed FB networks, while the middleware representation, runtime-associated orchestrators, service registries, bindings, and dashboard provide persistent representation, discovery, composition, monitoring, and coordination.

The evaluation confirms that this separation is useful, but also shows that it introduces a clear trade-off. The proposed workflow reduces manual configuration effort and provides a more integrated orchestration process, but it also adds processes, middleware communication, state synchronization, monitoring traffic, and memory consumption. Therefore, the contribution should not be interpreted as an unconditional performance improvement over the traditional DINASORE workflow. It is more accurately understood as an improvement in abstraction, visibility, service discovery, composition, and orchestration support, achieved at the cost of additional resource usage.

The runtime-overhead results in Section 5.2.1 illustrate this trade-off. The proposed workflow reduced the end-to-end time required to configure and connect the evaluated application, but it consumed more CPU and RAM than the traditional workflow. This result is consistent with the architectural decision to add an orchestration layer around the runtime instead of modifying the runtime itself. The additional CPU and memory usage are not accidental effects, but consequences

of maintaining persistent runtime state, synchronizing middleware information, processing MQTT and HTTP messages, and coordinating bindings through runtime-associated orchestrators.

The workflow and usability results in Section 5.2.2 show the practical value of this additional layer. Participants completed the evaluated task faster with the proposed workflow, and the questionnaire results indicate positive perceptions regarding monitoring, instance management, service discovery, binding creation, and learning effort. These results suggest that the dashboard and middleware representation made the orchestration process easier to understand and execute in the evaluated scenario. At the same time, the qualitative feedback shows that the dashboard remains a prototype. Missing feedback, incomplete navigation, and limited functionality still affect the usability of the interface and prevent claims about maturity as an engineering environment.

The scalability results in Section 5.2.3 show that the prototype remained operational up to the completed 500-instance configurations, while the 600-instance configurations failed when the Raspberry Pi devices approached resource exhaustion. This result is important because it identifies the edge devices, rather than the local middleware host, as the first limiting components in the tested setup. The result does not define a general upper bound for the architecture, DINASORE, Eclipse Ditto, or MQTT. It shows that, in the implemented prototype and hardware configuration, orchestration responsibilities placed close to runtime devices consume enough resources to make edge capacity a central scalability concern.

The use of Python scripts in the scalability procedure described in Section 5.1.4 also affects the interpretation of the results. These scripts emulated the message exchange normally produced by the dashboard when interacting with Eclipse Ditto, allowing the test environment to be constructed automatically and repeatedly. This made it possible to evaluate large numbers of runtime instances and bindings without manual interaction. However, it also means that the scalability tests evaluate the orchestration backend, middleware communication path, and runtime-associated orchestrators under dashboard-equivalent commands, not the rendering performance or interaction latency of the dashboard interface itself.

From an architectural perspective, the results support the separation between the execution plane and the orchestration plane. Local IEC 61499 execution remains inside DINASORE, while orchestration operations are represented through persistent middleware state and translated by runtime-associated orchestrators. This separation helps preserve the role of the runtime and avoids embedding middleware behavior inside the FB execution semantics. At the same time, it introduces dependencies on the middleware, MQTT broker, orchestrators, and dashboard bridge for discovery, synchronization, binding management, and visibility.

The results also show that service discovery depends strongly on the quality and availability of metadata. The distinction between runtime capabilities and deployed services is important: the FB registry describes what a DINASORE instance can instantiate, while the SOA registry describes which deployed FB instances are available as services. This distinction allows the orchestration workflow to reason about available services rather than only about possible FB types. Without persistent service descriptors, discovery would remain a manual inspection activity rather than a mechanism for composition.

The binding mechanism is another important outcome. By representing bindings as persistent orchestration entities, the proposed workflow makes cross-pipeline communication visible and configurable outside the application model. This reduces the need to embed explicit communication FBs into every application structure. However, it also introduces new responsibilities, such as validating compatibility between source and target variables, managing binding lifecycle, detecting stale bindings, and handling failures in the communication path. These responsibilities were only partially addressed in the prototype.

Overall, the evaluation supports the feasibility of the proposed orchestration workflow within the tested scope. It demonstrates that DINASORE-based IEC 61499 applications can be represented, discovered, connected, and monitored through a persistent service-oriented layer. The results also make clear that such a layer is not free: it introduces overhead, increases system complexity, and depends on the robustness of the middleware and orchestration components

6.2 Applicability, Benefits, and Costs

The proposed architecture is most appropriate for IEC 61499 systems in which several runtime instances, pipelines, and deployed FB instances need to be coordinated after deployment. It is particularly relevant when engineers need to discover available functionality, inspect runtime state, reuse already deployed FB instances, or connect independently deployed pipelines without manually embedding all communication details inside each application. In these situations, the value of the architecture comes from persistent representation, service discovery, dashboard-supported orchestration, and explicit binding management.

The architecture is also appropriate when orchestration activities are not part of hard real-time control loops. Runtime discovery, pipeline deployment, monitoring, service registration, and binding configuration are management-plane activities. They can tolerate the additional communication and synchronization introduced by the middleware layer more easily than time-critical control logic. For this reason, the proposed approach is better interpreted as an orchestration and coordination layer around IEC 61499 execution, rather than as a replacement for deterministic runtime communication.

The architecture is less suitable for small, static, or single-runtime systems where the application structure rarely changes and where the overhead of a middleware-supported orchestration layer would not be justified. In such cases, the traditional DINASORE workflow may be sufficient because the added visibility, discovery, and binding mechanisms would bring limited practical benefit. The approach is also less suitable for communication paths that require strict timing guarantees, since the prototype routes orchestration and binding operations through middleware, MQTT communication, and local management requests.

The main benefits observed in the evaluation were increased visibility, reduced manual configuration effort, service-oriented abstraction of deployed FB instances, and explicit representation of bindings between pipelines. The middleware representation made runtimes, pipelines, services, and bindings observable beyond the local runtime process. The dashboard integrated several tasks

that would otherwise be distributed across engineering tools, configuration files, and manual communication setup. The binding mechanism also made cross-pipeline communication visible at the orchestration level, instead of hiding it inside application-specific communication FBs.

These benefits are accompanied by clear costs. The architecture introduces additional processes, middleware dependencies, MQTT and HTTP communication, persistent state synchronization, and local orchestration logic. These elements increase CPU and RAM usage, add network traffic, and create additional failure points. The system also becomes more complex to configure, monitor, secure, and maintain. Therefore, the proposed workflow is most justified when the benefits of visibility, discovery, reuse, and managed composition outweigh the additional infrastructure and runtime overhead.

6.3 Answers to the Research Questions

RQ1: How can service-oriented principles be integrated into an orchestration layer around IEC 61499 runtime instances while preserving local FB execution?

The work shows that service-oriented principles can be integrated around IEC 61499 runtime instances by applying them to the orchestration plane rather than to the internal execution semantics of the runtime. DINASORE continues to execute FB networks locally, while the surrounding orchestration layer represents runtimes, deployed pipelines, services, and bindings through middleware state.

In the implemented prototype, each DINASORE instance is associated with an orchestrator that observes middleware updates, translates them into DINASORE-compatible XML management commands, and publishes runtime information back to the persistent representation. This allows deployed FB instances to be described as services without modifying the local execution model of DINASORE. Service orientation is therefore used to structure discovery, abstraction, composition, and orchestration around the runtime, not to replace the runtime.

This approach preserves the separation between control execution and orchestration. Time-critical FB execution remains inside the runtime, while service registration, service discovery, monitoring, and binding management are handled externally. This separation is important because the evaluated middleware and binding mechanisms were not designed or validated as hard real-time control mechanisms.

RQ2: What architectural mechanisms are required to support persistent runtime representation, service discovery, service composition, and inter-pipeline communication in IEC 61499-based distributed control systems?

The work indicates that four mechanisms are central. The first is a persistent information model, defined in Section 3.3 and implemented in Section 4.2. This model represents runtime instances, deployed pipelines, available FB types, service descriptors, runtime state, and bindings. Without this persistent representation, the orchestration workflow would depend on the transient state of the dashboard or on manual inspection of individual runtime instances.

The second mechanism is a runtime-associated orchestration boundary. The orchestrator described in Section 4.3 connects a local DINASORE instance to the middleware representation. It handles runtime communication, service and binding state, monitoring information, and the translation between middleware-level updates and DINASORE management commands.

The third mechanism is service registration and discovery. The implementation described in Section 4.4 separates deployable FB capabilities from deployed service instances. This distinction allows the workflow to identify services that are currently available and bindable, rather than only listing FB types that a runtime could instantiate.

The fourth mechanism is a persistent binding representation. Bindings describe relationships between independently deployed pipelines by associating a source output, a target input, and a target event. This mechanism supports inter-pipeline communication without requiring explicit communication FBs to be inserted manually into every application model.

RQ3: What practical benefits, limitations, and runtime overheads emerge when the proposed service-oriented orchestration workflow is compared with the traditional DINASORE workflow?

The main practical benefit is workflow integration. The proposed workflow brings runtime selection, pipeline deployment, service discovery, binding creation, and monitoring into a unified orchestration process. In the evaluated scenario, this reduced the time required to configure and connect the distributed application and reduced the number of explicit manual steps required from participants.

The workflow also improved abstraction. Instead of requiring the user to manually connect pipelines through explicit MQTT publisher and subscriber FBs, communication could be configured as a binding between services. This made cross-pipeline communication visible at the orchestration level and allowed independently deployed pipelines to be connected through persistent middleware state.

The limitations are also clear. The orchestration layer increased CPU and RAM usage, introduced dependency on middleware services, and added communication paths that were not evaluated for deterministic timing. The scalability tests showed that the edge devices became limiting components in the tested setup, especially when many runtime instances and bindings were active. The participant study also showed that the dashboard needs further development before it can be considered a mature engineering tool.

Therefore, the proposed workflow should be interpreted as a practical improvement in orchestration and configuration support for the evaluated scenario, not as proof of general industrial interoperability, general reusability, general reconfiguration capability, or improved deterministic performance. Its main value is to demonstrate that a persistent service-oriented orchestration layer can be added around DINASORE-based IEC 61499 applications, while making the associated costs and limitations visible.

6.4 Limitations

The limitations of this work are related to the evaluation scope, the maturity of the prototype, and the technical aspects that were not fully characterized. The evaluation was performed in a controlled laboratory environment and with a small participant sample. The participants were mostly academic users, so the results cannot be interpreted as evidence of industrial usability or adoption readiness. The evaluated scenario was also limited to a specific application structure, which restricts the generalization of the observed usability and engineering-effort results.

The results are also tied to the concrete prototype used in the evaluation. DINASORE was used as the IEC 61499 RTE case study, while Eclipse Ditto, Eclipse Mosquitto, Python orchestrators, and a dashboard bridge were used to instantiate the proposed architecture. Although the architectural principles can be considered beyond DINASORE, the evaluation does not prove that the same behavior, overhead, or usability would be obtained with other IEC 61499 RTEs or middleware platforms. Additional implementations would be required to validate the portability of the approach.

The prototype does not fully characterize timing and dependability properties. The evaluation measured execution overhead, workflow time, participant feedback, and scalability indicators, but it did not measure binding latency, jitter, message ordering, duplicated messages, recovery after middleware failures, long-running stability, or behavior under network degradation. These aspects are important for industrial systems and would need to be evaluated before the approach could be considered for production settings.

Security and resilience were also outside the implemented scope. The prototype does not provide a complete treatment of authentication, authorization, encrypted communication, credential management, topic-level access control, broker failover, middleware replication, or systematic recovery after partial failures. These limitations do not invalidate the architectural contribution, but they restrict the conclusions that can be drawn about production deployment.

Finally, the dashboard remains a prototype interface. It was sufficient to evaluate the proposed workflow, but it does not yet provide the validation, feedback, error handling, lifecycle-management support, and user guidance expected from a mature engineering tool. Participant feedback also indicated that some aspects of the interface and available functionality require refinement. For this reason, the dashboard should be understood as an evaluation artifact rather than as a production-ready engineering environment.

Chapter 7

Conclusions

This dissertation investigated how service-oriented principles can be used to support the orchestration of distributed IEC 61499 applications executed by runtime environments at the edge. The starting point was the problem defined in Section 1.3: IEC 61499 provides a model for distributed and event-driven control applications, but distributed execution alone does not provide a persistent orchestration workflow for representing runtime instances, deployed pipelines, available capabilities, service descriptors, and bindings. The work therefore proposed an edge-centric service-oriented orchestration architecture that preserves local FB execution inside the RTE while adding a middleware-supported layer for representation, discovery, monitoring, composition, and cross-pipeline coordination.

DINASORE was used as the concrete IEC 61499 RTE case study through which the architecture was implemented and evaluated. This choice made it possible to validate the proposed workflow in a real execution environment while keeping the architectural contribution broader than a single implementation. DINASORE remained responsible for executing deployed FB networks and interpreting IEC 61499 management commands, while the orchestration layer maintained runtime state, pipeline representations, service information, and binding definitions outside the local runtime process.

The work combined three main directions. IEC 61499 provided the execution and application model, SOA provided the principles of abstraction, loose coupling, discovery, and composition, and middleware-supported DT representation provided the persistent state required to coordinate distributed runtime instances. The resulting architecture separates execution concerns from orchestration concerns. Local FB execution remains inside the RTE, while the service-oriented layer represents the distributed system, exposes deployed FB instances as services, supports discovery, and manages bindings between independently deployed pipelines.

The prototype demonstrated how this architecture can be instantiated using DINASORE, Eclipse Ditto, Eclipse Mosquitto, runtime-associated Python orchestrators, service and binding mechanisms, and a dashboard interface. Eclipse Ditto provided the persistent representation of runtime instances, pipelines, services, and bindings. Eclipse Mosquitto supported asynchronous communication between the middleware, orchestrators, and runtime-associated components. The orches-

trators translated middleware-level changes into DINASORE-compatible management requests, while the dashboard integrated runtime discovery, pipeline deployment, monitoring, service discovery, and binding configuration into a single workflow.

The evaluation showed that the proposed workflow can provide practical benefits in the evaluated scenario. Compared with the traditional DINASORE workflow, the proposed approach improved the visibility of runtime instances, deployed pipelines, available services, and binding relationships. The participant study also indicated that the dashboard-supported workflow reduced configuration effort for the evaluated task and was perceived positively for monitoring, instance management, service discovery, binding creation, and learning effort. These results support the feasibility of service-oriented orchestration around IEC 61499 runtime instances, especially when systems involve multiple runtimes, independently deployed pipelines, and functionality that should be discovered or reused after deployment.

At the same time, the evaluation also showed that these benefits are not obtained without cost. The proposed workflow introduced additional CPU and RAM usage, additional communication through MQTT and HTTP, and a dependency on middleware services for orchestration-plane functionality. The scalability tests showed that resource-constrained edge devices can become limiting components as the number of runtime instances and bindings increases. Therefore, the contribution should be interpreted as a trade-off rather than as an unconditional improvement over the traditional workflow. The architecture improves abstraction, discoverability, observability, and managed composition in the evaluated context, but it also increases infrastructure complexity and runtime overhead.

7.1 Contributions

The contributions of this dissertation can be understood at two complementary levels: the architectural contribution and the implementation and evaluation artifacts used to validate it. At the architectural level, the work defined an edge-centric service-oriented orchestration approach for IEC 61499 distributed control systems. The architecture separates local FB execution from orchestration concerns and shows how runtime instances, deployed pipelines, runtime capabilities, service descriptors, and bindings can be coordinated through a middleware-supported service-oriented layer.

A central contribution is the persistent information model that supports this orchestration workflow. The model distinguishes between the FB types that a runtime can instantiate, the pipelines that are currently deployed, the FB instances that are exposed as services, and the bindings that connect independently deployed pipelines. This distinction is important because service discovery and service composition require more than access to the original application structure: they require a persistent representation of what exists at runtime, where it is deployed, and how it can be connected.

The dissertation also contributes a service-abstraction mechanism for deployed FB instances. In the proposed workflow, an FB instance deployed inside an IEC 61499 runtime can be represented as a service descriptor containing its pipeline, FB type, inputs, outputs, events, runtime location, and availability. This makes deployed functionality discoverable beyond the local runtime process, while preserving the execution responsibilities of the IEC 61499 RTE.

Another contribution is the binding model for inter-pipeline communication. The binding mechanism represents the association between source services, target services, variables, and trigger events as an explicit orchestration entity. This allows independently deployed pipelines to remain separate while still supporting managed value propagation between them. As a result, cross-pipeline communication becomes visible and configurable at the orchestration level instead of being hidden only inside application-specific communication logic.

These architectural contributions were validated through a DINASORE-based prototype implementation. The prototype instantiates the architecture using Eclipse Ditto, Eclipse Mosquitto, runtime-associated Python orchestrators, service and binding mechanisms, and a dashboard interface. The implementation demonstrates that the proposed concepts can be integrated around an existing IEC 61499 RTE without replacing its local execution model.

The evaluation provides the final contribution by characterizing the practical benefits and costs of the proposed workflow. The results show that the architecture can improve visibility, service discovery, binding configuration, and workflow integration in the evaluated scenario, while also introducing additional CPU usage, memory usage, communication overhead, middleware dependency, and scalability constraints. This analysis clarifies when the approach is useful, which trade-offs it introduces, and which limitations remain before broader industrial use can be considered.

The work also contributed to research dissemination. Part of the research developed during this dissertation resulted in a peer-reviewed publication at CLOSER 2025 [44]. This publication was associated with the SOA and literature-review component of the dissertation, focusing on cloud-based service-oriented approaches for IEC 61499 distributed control systems and helping to position the architectural gap addressed by the proposed work. A second output of this work is the accepted paper *Enabling Human-Centric Cyber-Physical Systems through Service-Oriented IEC 61499 Edge Architectures* [45]. This paper is associated with the service-oriented architecture and prototype workflow developed in this dissertation, focusing on the use of SOA principles, edge orchestration, and IEC 61499 runtime integration to support more human-centric management of distributed cyber-physical systems. It is expected to be presented at ICIEA 2026, the 21st IEEE Conference on Industrial Electronics and Applications, in August 2026.

The conclusions of this work must be interpreted together with the limitations discussed in Section 6.4. The evaluation was exploratory, the participant sample was small and mostly academic, the validation scenario was limited, and the scalability tests were conducted in a controlled laboratory setup. The prototype also did not characterize deterministic timing, binding latency, jitter, message ordering, security, resilience, or long-running behavior. As a result, the dissertation supports the feasibility and practical usefulness of the proposed workflow in the tested context, but it does not prove industrial readiness, production-level scalability, or suitability for hard real-time

communication paths.

Future development should first focus on reducing the overhead of the orchestration layer. This can include replacing unnecessary polling with event-driven updates, batching registry and monitoring operations, reducing redundant middleware writes, and investigating alternative deployment strategies for the orchestrator. In the current prototype, each DINASORE runtime is associated with a local orchestrator executing on the same edge device. Future implementations could evaluate a dedicated orchestration layer deployed independently of the runtime instances, centralizing orchestration responsibilities while preserving the distributed execution of IEC 61499 applications. Such changes could reduce CPU, RAM, and network usage on resource-constrained edge devices while simplifying orchestration management.

The dashboard also requires further refinement. The participant feedback indicates that the prototype interface was useful for evaluating the workflow, but it still needs stronger validation, clearer deployment feedback, better error handling, more direct service navigation, and closer integration between pipeline editing, monitoring, discovery, and binding configuration. A mature engineering interface should also support more explicit guidance when bindings are invalid, when runtime information is stale, or when deployment operations fail.

The service and binding models could also be extended with richer metadata. Information about variable types, event compatibility, lifecycle state, runtime location, versioning, expected latency, and reliability requirements would allow the system to validate bindings before they are created and improve the usefulness of service discovery in larger deployments. Such metadata would also make the architecture more suitable for heterogeneous IEC 61499 environments beyond the DINASORE case study.

Further evaluation is also necessary. Future studies should include longer execution periods, additional application scenarios, different IEC 61499 RTEs, different edge hardware, controlled network impairments, and participants with industrial automation experience. Measurements such as binding latency, jitter, message ordering, recovery after middleware failures, long-term memory behavior, and behavior under network degradation would be required to characterize the architecture beyond the exploratory evaluation presented in this dissertation.

Finally, security and resilience must be addressed before considering industrial deployment. A production-oriented version of the architecture would require authentication, authorization, encrypted communication, credential management, topic-level access control, middleware replication, broker failover, reconnect behavior, stale-entry cleanup, and recovery mechanisms after partial failures.

Overall, the dissertation shows that service-oriented orchestration can be introduced around IEC 61499 runtime instances while preserving local FB execution. Within the evaluated scope, the proposed workflow demonstrates the practical value of persistent runtime representation, service discovery, dashboard-supported deployment, and explicit cross-pipeline binding. At the same time, the work makes clear that broader adoption depends on reducing overhead, improving the dashboard, strengthening validation, and evaluating the architecture in more realistic industrial conditions.

References

[1] Virendra Ashiwal, Mainak Majumder, and Alois Zoitl. "Evaluation of Middleware Technologies for the PLC-Service Bus in IEC 61499". In: *2022 IEEE 27th International Conference on Emerging Technologies and Factory Automation (ETFA)*. 2022, pp. 1–4. DOI: 10.1109/ETFA52439.2022.9921536.

[2] Virendra Ashiwal et al. "Apache Kafka as a Middleware to Support the PLC-Service Bus Architecture with IEC 61499". In: *Software Architecture. ECSA 2022 Tracks and Workshops*. Ed. by Thais Batista et al. Cham: Springer International Publishing, 2023, pp. 62–74. ISBN: 978-3-031-36889-9.

[3] Virendra Ashiwal et al. "Identification and Evaluation of Pitfalls in the Migration From IEC 61131-3 to IEC 61499: A Review". In: *IEEE Open Journal of the Industrial Electronics Society* 6 (2025), pp. 575–590. DOI: 10.1109/OJIES.2025.3558685.

[4] Abdullah Aziz et al. "Empowering The Eclipse Arrowhead Framework with a Digital Twin as a Proxy Service". In: *2022 22nd International Conference on Control, Automation and Systems (ICCAS)*. 2022, pp. 1716–1721. DOI: 10.23919/ICCAS55662.2022.10003919.

[5] Zakaria Bencherki et al. "The landscape of IEC 61499 in modern automation: A scoping review of research trends and gaps (2018–2025)". In: *Journal of Process Control* 162 (2026), p. 103726. ISSN: 0959-1524. DOI: https://doi.org/10.1016/j.jprocont.2026.103726. URL: https://www.sciencedirect.com/science/article/pii/S0959152426001095.

[6] David Albert Breunig, Alain Roedel, and Thomas Bauernhansl. "Extending Service-oriented Architectures in Manufacturing towards Fog and Edge Levels". In: *2020 IEEE 18th International Conference on Industrial Informatics (INDIN)*. Vol. 1. 2020, pp. 291–298. DOI: 10.1109/INDIN45582.2020.9442152.

[7] James Christensen et al. "The IEC 61499 Function Block Standard: Overview of the Second Edition". In: Jan. 2012.

[8] Alexandre Rosas Costa, Luisa Jorge, and Paulo Melo. "Open Source Platforms for Digital Twin Development: Implementation of a Simplified Case Study". In: *2025 12th International Conference on Internet of Things: Systems, Management and Security (IOTSMS)*. 2025, pp. 101–106. DOI: 10.1109/IOTSMS68530.2025.11408468.

[9] Wenbin Dai, Wanqi Huang, and Valeriy Vyatkin. "Enabling plug-and-play software components in industrial cyber-physical systems by adopting service-oriented architecture paradigm". In: *IECON 2016 - 42nd Annual Conference of the IEEE Industrial Electronics Society*. 2016, pp. 5253–5258. DOI: 10.1109/IECON.2016.7793834.

[10] Wenbin Dai et al. "Bridging Service-Oriented Architecture and IEC 61499 for Flexibility and Interoperability". In: *IEEE Transactions on Industrial Informatics* 11.3 (2015), pp. 771–781. DOI: 10.1109/TII.2015.2423495.

[11] Wenbin Dai et al. “Service-oriented distributed control software design for process automation systems”. In: *2014 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*. 2014, pp. 3637–3642. DOI: 10.1109/SMC.2014.6974495.

[12] Wenbin William Dai, Valeriy Vyatkin, and James H. Christensen. “The application of service-oriented architectures in distributed automation systems”. In: *2014 IEEE International Conference on Robotics and Automation (ICRA)*. 2014, pp. 252–257. DOI: 10.1109/ICRA.2014.6906618.

[13] Wenbin William Dai et al. “Function block implementation of service oriented architecture: Case study”. In: *2014 12th IEEE International Conference on Industrial Informatics (INDIN)*. 2014, pp. 112–117. DOI: 10.1109/INDIN.2014.6945493.

[14] Ernst Dersch et al. “Arrowhead Framework: A Systems View on Service Orientation for Industrial Automation”. In: *IEEE Transactions on Industrial Informatics* 13.2 (2017), pp. 1000–1010. DOI: 10.1109/TII.2016.2631939.

[15] DIGI2-FEUP. *3.2. Hands-On: Distributed Sensorization*. DINASORE Wiki. 2021. URL: https://github.com/DIGI2-FEUP/dinasore/wiki/3.2.-Hands-On:-Distributed-Sensorization (visited on 06/08/2026).

[16] DIGI2-FEUP. *DINASORE: Distributed Industrial Automation and Control System*. https://github.com/DIGI2-FEUP/dinasore. Accessed: 2024-07-12. 2024.

[17] *Eclipse Ditto – Digital Twins for IoT and Connected Devices*. https://www.eclipse.org/ditto/. Accessed: 2026-01-xx. Eclipse Foundation, 2026.

[18] Eclipse Foundation. *Eclipse 4diac Framework*. Accessed: 2026-05-30. 2026. URL: https://eclipse.dev/4diac/doc/intro/4diacframework.html.

[19] L. Ferrarini and C. Veber. “Implementation approaches for the execution model of IEC 61499 applications”. In: *2nd IEEE International Conference on Industrial Informatics, 2004. INDIN ’04. 2004*. 2004, pp. 612–617. DOI: 10.1109/INDIN.2004.1417418.

[20] Michele Foletti et al. “Implementation and System Architecture Challenges for Event-Based Programmable Logic Controllers - A Literature Review”. In: *Procedia CIRP* 130 (2024). 57th CIRP Conference on Manufacturing Systems 2024 (CMS 2024), pp. 1529–1536. ISSN: 2212-8271. DOI: https://doi.org/10.1016/j.procir.2024.10.278. URL: https://www.sciencedirect.com/science/article/pii/S2212827124014355.

[21] Priyanshi Gangrade et al. “Edge and Fog Computing in Cyber-Physical Systems”. In: *2025 International Conference on Intelligent Control, Computing and Communications (IC3)*. 2025, pp. 172–176. DOI: 10.1109/IC363308.2025.10956222.

[22] Roger Tadeu Giglio et al. “Orquestração de Microsserviços usando IEC 61499 para Controle de Processos na Indústria 4.0”. In: *XXV Congresso Brasileiro de Automática (CBA 2024)*. Brasil: Sociedade Brasileira de Automática (SBA), 2024, pp. 3363–3368. DOI: 10.20906/CBA2024/4670.

[23] Peter Gsellmann et al. “A Novel Approach for Integrating IEC 61131-3 Engineering and Execution Into IEC 61499”. In: *IEEE Transactions on Industrial Informatics* 17.8 (2021), pp. 5411–5418. DOI: 10.1109/TII.2020.3033330.

[24] David Hästbacka et al. “Dynamic Edge and Cloud Service Integration for Industrial IoT and Production Monitoring Applications of Industrial Cyber-Physical Systems”. In: *IEEE Transactions on Industrial Informatics* 18.1 (2022), pp. 498–508. DOI: 10.1109/TII.2021.3071509.

[25] Florian Held, Philipp Schauz, and Jörg Domaschka. "A Systematic Comparison of IoT Middleware". In: *Service-Oriented and Cloud Computing*. Ed. by Fabrizio Montesi, George Angelos Papadopoulos, and Wolf Zimmermann. Cham: Springer International Publishing, 2022, pp. 77–92. ISBN: 978-3-031-04718-3.

[26] Pranay Jhunjhunwala, Udayanto Dwi Atmojo, and Valeriy Vyatkin. "Towards Implementation of Interoperable Smart Sensor Services in IEC 61499 for Process Automation". In: *2020 25th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. Vol. 1. 2020, pp. 1409–1412. DOI: 10.1109/ETFA46521.2020.9211925.

[27] Guolin Lyu and Robert William Brennan. "Towards IEC 61499-Based Distributed Intelligent Automation: A Literature Review". In: *IEEE Transactions on Industrial Informatics* 17.4 (2021), pp. 2295–2306. DOI: 10.1109/TII.2020.3016990.

[28] Rodolfo Medeiros, Sílvio Fernandes, and Paulo G. G Queiroz. "Middleware for the Internet of Things: a systematic literature review". In: *JUCS - Journal of Universal Computer Science* 28.1 (2022), pp. 54–79. ISSN: 0948-695X. DOI: 10.3897/jucs.71693. eprint: https://doi.org/10.3897/jucs.71693. URL: https://doi.org/10.3897/jucs.71693.

[29] László Monostori. "Cyber-physical Production Systems: Roots, Expectations and RD Challenges". In: *Procedia CIRP* 17 (2014). Variety Management in Manufacturing, pp. 9–13. ISSN: 2212-8271. DOI: https://doi.org/10.1016/j.procir.2014.03.115. URL: https://www.sciencedirect.com/science/article/pii/S2212827114003497.

[30] Jeff Morgan et al. "Industry 4.0 smart reconfigurable manufacturing machines". In: *Journal of Manufacturing Systems* 59 (2021), pp. 481–506. ISSN: 0278-6125. DOI: https://doi.org/10.1016/j.jmsy.2021.03.001. URL: https://www.sciencedirect.com/science/article/pii/S027861252100056X.

[31] Alessia Napoleone et al. "How the technologies underlying cyber-physical systems support the reconfigurability capability in manufacturing: a literature review". In: *International Journal of Production Research* 61.9 (2023), pp. 3122–3144. DOI: 10.1080/00207543.2022.2074323. eprint: https://doi.org/10.1080/00207543.2022.2074323. URL: https://doi.org/10.1080/00207543.2022.2074323.

[32] Kanu Patel and Hardik Modi. "Analysis and Review on Service-Oriented-Based IoT Middleware". In: *Emerging Technologies in Data Mining and Information Security*. Ed. by João Manuel R. S. Tavares et al. Singapore: Springer Singapore, 2021, pp. 197–209. ISBN: 978-981-15-9774-9.

[33] E. M. Pereira, J. P. C. dos Reis, and G. Gonçalves. "DINASORE: A Dynamic Intelligent Reconfiguration Tool for Cyber-Physical Production Systems". In: *SAM IoT*. 2020, pp. 63–71.

[34] Ricardo Pasquati Pontarolli et al. "Microservice-Oriented Architecture for Industry 4.0". In: *Eng* 4.2 (2023), pp. 1179–1197. ISSN: 2673-4117. DOI: 10.3390/eng4020069. URL: https://www.mdpi.com/2673-4117/4/2/69.

[35] Laurin Prenzel, Alois Zoitl, and Julien Provost. "IEC 61499 Runtime Environments: A State of the Art Comparison". In: *Computer Aided Systems Theory – EUROCAST 2019*. Ed. by Roberto Moreno-Díaz, Franz Pichler, and Alexis Quesada-Arencibia. Cham: Springer International Publishing, 2020, pp. 453–460. ISBN: 978-3-030-45096-0.

[36] Laurin Prenzel, Alois Zoitl, and Julien Provost. “Run-time Configuration of the IEC 61499-based PLC Service Bus via OPC UA”. In: *Proceedings of the 16th International Conference on Industrial Informatics (INDIN)*. 2020, pp. 453–460. DOI: 10.1007/978-3-030-45096-0_55.

[37] Reagraph. *Reagraph Documentation*. 2026. URL: https://reagraph.dev/docs (visited on 06/21/2026).

[38] Jacqueline Zonichenn Reis and Rodrigo Franco Gonçalves. “The Role of Internet of Services (IoS) on Industry 4.0 Through the Service Oriented Architecture (SOA)”. In: *Advances in Production Management Systems. Smart Manufacturing for Industry 4.0*. Ed. by Ilkyeong Moon et al. Cham: Springer International Publishing, 2018, pp. 20–26. ISBN: 978-3-319-99707-0.

[39] Daniel Schel et al. “Manufacturing Service Bus: An Implementation”. In: *Procedia CIRP* 67 (2018). 11th CIRP Conference on Intelligent Computation in Manufacturing Engineering, 19-21 July 2017, Gulf of Naples, Italy, pp. 179–184. ISSN: 2212-8271. DOI: https://doi.org/10.1016/j.procir.2017.12.196. URL: https://www.sciencedirect.com/science/article/pii/S221282711731140X.

[40] Frank Siqueira and Joseph G. Davis. “Service Computing for Industry 4.0: State of the Art, Challenges, and Research Opportunities”. In: *ACM Comput. Surv.* 54.9 (Oct. 2021). ISSN: 0360-0300. DOI: 10.1145/3478680. URL: https://doi.org/10.1145/3478680.

[41] Thomas Strasser et al. “Framework for Distributed Industrial Automation and Control (4DIAC)”. In: *2008 6th IEEE International Conference on Industrial Informatics*. 2008, pp. 283–288. DOI: 10.1109/INDIN.2008.4618110.

[42] Miglena Temelkova. “Similarities and Differences Between the Technological Paradigms "Production System", "Cyber-physical System" and "Cyber-physical Production System"”. In: *2022 International Conference on Communications, Information, Electronic and Energy Systems (CIEES)*. 2022, pp. 1–7. DOI: 10.1109/CIEES55704.2022.9990698.

[43] K. Thramboulidis. “Service-oriented architecture in industrial automation systems—the case of IEC 61499: A review”. In: *arXiv preprint* arXiv:1506.04615 (2015). arXiv preprint arXiv:1506.04615.

[44] Tomás Torres, Gil Gonçalves, and Rui Pinto. “Literature Review on Cloud-Based Service-Oriented Architecture for IEC 61499 Distributed Control Systems”. In: *Proceedings of the 15th International Conference on Cloud Computing and Services Science - CLOSER*. INSTICC. SciTePress, 2025, pp. 174–181. ISBN: 978-989-758-747-4. DOI: 10.5220/0013293400003950.

[45] Tomás Torres and Rui Pinto. “Enabling Human-Centric Cyber-Physical Systems through Service-Oriented IEC 61499 Edge Architectures”. Accepted for presentation at the 21st IEEE Conference on Industrial Electronics and Applications (ICIEA 2026), Catania, Italy, 02–06 August 2026. 2026. URL: https://www.ieeeiciea.org/2026/.

[46] Valeriy Vyatkin. “IEC 61499 as enabler of distributed and intelligent automation: State-of-the-art review”. In: *IEEE Transactions on Industrial Informatics* 7.4 (2011). Cited by: 325, pp. 768–781. DOI: 10.1109/TII.2011.2166785. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-81255138946&doi=10.1109%2fTII.2011.2166785&partnerID=40&md5=c16c798d4dd393e7695b3c22d2160ba7.

[47] xyflow. *React Flow: Node-Based UIs in React*. 2026. URL: https://reactflow.dev/ (visited on 06/21/2026).

[48] Mingjin Zhang et al. “EaaS: A Service-Oriented Edge Computing Framework Towards Distributed Intelligence”. In: *2022 IEEE International Conference on Service-Oriented System Engineering (SOSE)*. 2022, pp. 165–175. DOI: 10.1109/SOSE55356.2022.00026.

[49] Nan Zhou and Di Li. “Hybrid Synchronous–Asynchronous Execution of Reconfigurable PLC Programs in Edge Computing”. In: *IEEE Transactions on Industrial Informatics* 18.3 (2022), pp. 1663–1673. DOI: 10.1109/TII.2021.3092741.

[50] Alois Zoitl, Thomas Strasser, and Antonio Valentini. “Open source initiatives as basis for the establishment of new technologies in industrial automation: 4DIAC a case study”. In: *2010 IEEE International Symposium on Industrial Electronics*. 2010, pp. 3817–3819. DOI: 10.1109/ISIE.2010.5637502.

Appendix A

Eclipse Ditto Representation Examples

This appendix complements the mapping presented in Figure 4.1 by showing the concrete Eclipse Ditto representations used in the prototype. The examples are separated according to the three representation roles discussed in Chapter 4: the deployed pipeline Thing, the service descriptor stored in `soa:registry`, and the binding configuration used to connect services from independently deployed pipelines.

Listing A.1 shows how a DINASORE pipeline is represented as an Eclipse Ditto Thing. The Thing stores runtime-level attributes, deployed FB Features, event and variable values, and a dedicated `connections` Feature containing the internal IEC 61499 links used by the orchestrator during deployment and synchronization.

Listing A.1: Example of a DINASORE pipeline represented as an Eclipse Ditto Thing.

```
{
    "thingId": "dinasore:DIN1",
    "policyId": "my.test:policy",
    "attributes": {
        "description": "Digital twin of DINASORE EMB_RES resource",
        "status": "stopped",
        "type": "EMB_RES"
    },
    "features": {
        "SENSOR_SIMULATOR": {
            "definition": [
                "dinasore:DIN1:SENSOR_SIMULATOR"
            ],
            "properties": {
                "variables": {
                    "input": {
                        "OFFSET": 20
                    },
                    "output": {
                        "VALUE": ""
                    }
                },
                "events": {
```

```
                    "input": {
                        "INIT": 1,
                        "READ": 0
                    },
                    "output": {
                        "INIT_O": 0,
                        "READ_O": 0
                    }
                }
            }
        },
        "MOVING_AVERAGE_1": {
            "definition": [
                "dinasore:DIN1:MOVING_AVERAGE"
            ],
            "properties": {
                "variables": {
                    "input": {
                        "WINDOW": 10,
                        "VALUE": ""
                    },
                    "output": {
                        "VALUE_MA": ""
                    }
                },
                "events": {
                    "input": {
                        "INIT": 1,
                        "RUN": 0
                    },
                    "output": {
                        "INIT_O": 0,
                        "RUN_O": 0
                    }
                }
            }
        },
        "connections": {
            "definition": [
                "dinasore:DIN1:connections"
            ],
            "properties": {
                "links": [
                    {
                        "source": "SENSOR_SIMULATOR.VALUE",
                        "destination": "MOVING_AVERAGE_1.VALUE"
                    },
                    {
                        "source": "SENSOR_SIMULATOR.READ_O",
                        "destination": "MOVING_AVERAGE_1.RUN"
                    }
                ]
            }
        }
    }
}
```

Listing A.2 shows the service-oriented view of a deployed FB. In this representation, the FB is no longer shown as part of a complete pipeline structure, but as a discoverable service descriptor stored as a Feature inside `soa:registry`. The descriptor keeps the pipeline where the FB was deployed, the FB type, its service interface, and its availability state.

Listing A.2: Example of a service descriptor stored as a Feature inside `soa:registry`.

```
{
  "pipeline": "SENSOR_120",
  "fbType": "SENSOR_SIMULATOR",
  "inputs": {
    "OFFSET": "INT"
  },
  "outputs": {
    "VALUE": "INT"
  },
  "events": {
    "input": [
      "INIT",
      "READ"
    ],
    "output": [
      "INIT_O",
      "READ_O"
    ]
  },
  "status": "available"
}
```

Listing A.3 shows the configuration section of a binding Thing. The binding does not redefine the source or target FBs. Instead, it stores the mappings between source service outputs and target service inputs, together with the target event that must be triggered after the mapped values are written.

Listing A.3: Example of the configuration section of a binding Thing.

```
"configuration": {
    "properties": {
        "mappings": [
            {
                "from": {
                    "service": "DIN1:SENSOR_SIMULATOR",
                    "variable": "VALUE"
                },
                "to": {
                    "service": "DIN2:INFLUX_DB_2",
                    "variable": "VALUE_1"
                }
            },
            {
                "from": {
                    "service": "DIN1:MOVING_AVERAGE_1",
                    "variable": "VALUE_MA"
                },
                "to": {
                    "service": "DIN2:INFLUX_DB_2",
                    "variable": "VALUE_2"
                }
            }
        ],
        "trigger": {
            "service": "DIN2:INFLUX_DB_2",
            "event": "RUN"
        }
    }
}
```

Appendix B

Participant Study Protocol

This appendix summarizes the protocol used in the comparative usability study. The original instructions were presented in Portuguese, which was the language used with the participants. The protocol was adapted for inclusion in the dissertation by removing repeated interface instructions and retaining the information required to understand and reproduce the study.

B.1 Study Procedure

Each participant completed the same distributed sensorization scenario through two workflows: the traditional DINASORE workflow based on Eclipse 4diac and the proposed workflow based on Eclipse Ditto and the dashboard. Before each task, the corresponding DINASORE instances and supporting services were prepared by the researcher. Participants were asked to indicate when they started and when the timed task was complete.

The timed task ended after both pipelines had been deployed and communication between them had been configured. A short monitoring walkthrough was performed only after the timer had been stopped. This walkthrough demonstrated WATCH creation, event triggering, and value updates, but it was not considered a separate test stage and did not contribute to the completion time.

B.2 Traditional Workflow

Participants performed the following operations in Eclipse 4diac:

1. configure two DINASORE runtime devices and their management endpoints;
2. create the sensor-side and database-side FB networks;
3. configure the FB parameters listed in Table B.1;
4. add the required event and data connections;
5. map the FBs to the corresponding runtime devices;

6. add and configure `MQTT_PUBLISHER_2` and `MQTT_SUBSCRIBER_2` to connect the independent pipelines; and
7. deploy the complete application.

B.3 Proposed Workflow

Participants performed the corresponding operations through the dashboard:

1. select the sensor DINASORE instance and create the `SENSOR_SIMULATOR` and `MOVING_AVERAGE` pipeline;
2. configure the FB parameters and internal pipeline connections;
3. deploy the sensor pipeline;
4. select the database DINASORE instance and deploy `INFLUX_DB_2`;
5. discover the deployed database service;
6. create a binding between the sensor outputs and the database inputs; and
7. deploy the binding.

B.4 Common Application Parameters

Table B.1: Common Function Block parameters used in both workflows.

Function Block	**Configuration**
`SENSOR_SIMULATOR`	`OFFSET = 20`
`MOVING_AVERAGE`	`WINDOW = 10`
`INFLUX_DB_2`	Database host and port, credentials, database name, measurement name, and the two value names used for the sensor and moving-average values.
`MQTT_PUBLISHER_2` and `MQTT_SUBSCRIBER_2`	Common broker address, port, topic, and the two exchanged value names. These FBs were used only in the traditional workflow.

B.5 Monitoring Walkthrough

After completion time had been recorded, participants were shown how each interface exposed monitoring operations. In Eclipse 4diac, this involved enabling system monitoring, creating

WATCH operations, and triggering the relevant events. In the proposed dashboard, WATCH operations and event triggering were available from the FB configuration interface, while monitored values were displayed directly in the selected pipeline.

This walkthrough was included solely to support the monitoring-related questionnaire item. It was not timed and was not used as a separate performance or usability task.

B.6 Selected Interface Views

The complete dashboard layout used in the proposed workflow is presented in Figure 4.5. The appendix therefore retains only the task-specific views required to understand the participant procedure.

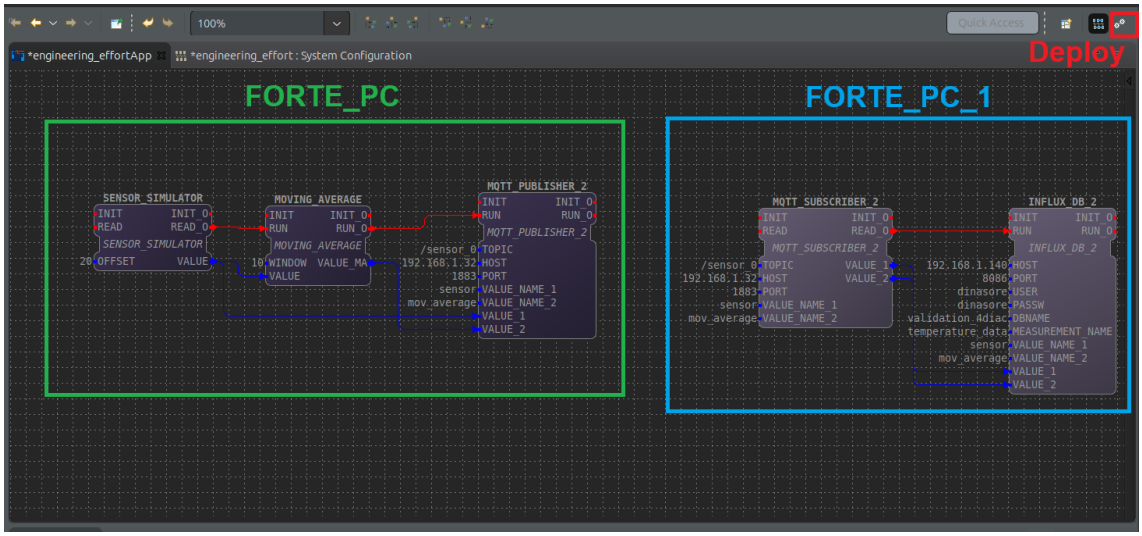


Figure B.1: Final distributed pipeline configured through Eclipse 4diac, including the FBs mapped to the two DINASORE runtime devices.

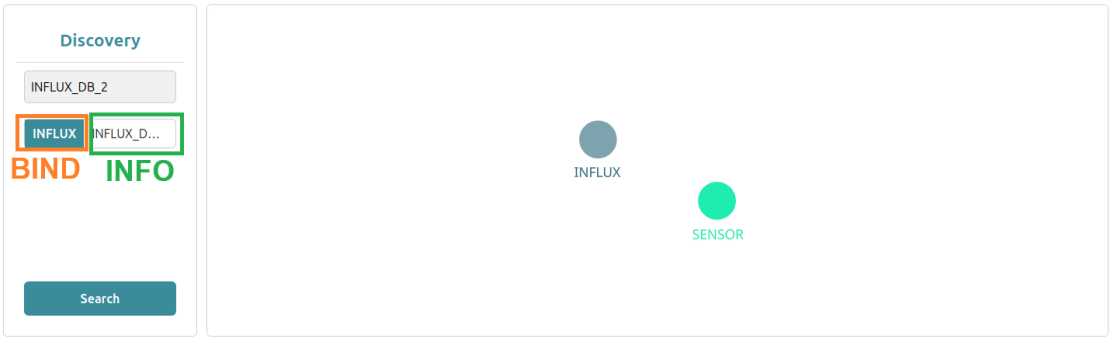


Figure B.2: Service discovery result and the binding created between the sensor and database DINASORE instances.

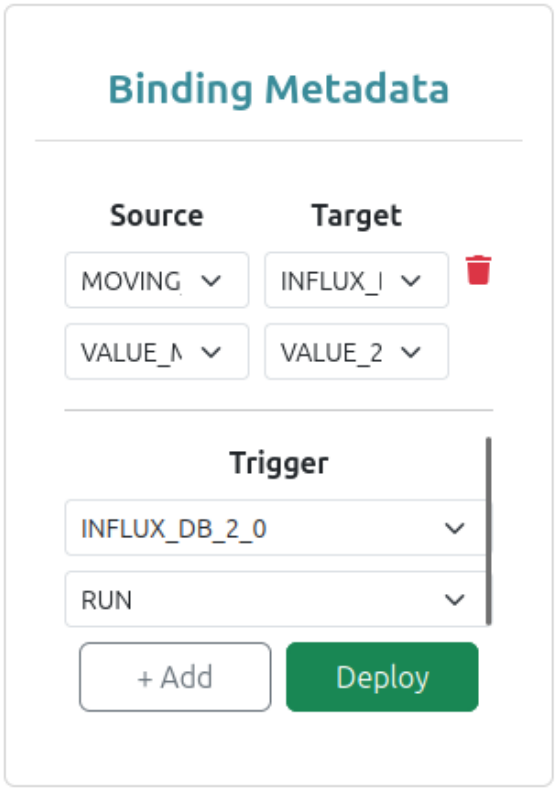


Figure B.3: Binding configuration used to associate the source outputs with the target inputs and trigger event.