FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

IEC 61499 Compliant Remote Web-based IDE for Function Block Design

Pedro da Cunha Teixeira e Faro Beirão

Mestrado em Engenharia Informática e Computação

Supervisor: Prof. Rui Pinto

July 26, 2026

IEC 61499 Compliant Remote Web-based IDE for Function Block Design

Pedro da Cunha Teixeira e Faro Beirão

Mestrado em Engenharia Informática e Computação

Approved in oral examination by the committee:

President: Prof. Konstantinos Bletsas
Examiner: Prof. José Barbosa

July 26, 2026

Resumo

Sistemas de Produção Ciber-Físicos (CPPSs) impõem exigências cada vez maiores às ferramentas de *software* utilizadas para desenhar e operar aplicações de controlo industrial distribuído. A norma *IEC 61499* fornece um modelo de blocos funcionais estruturado e orientado por eventos, adequado a estes ambientes, no entanto, as ferramentas de desenvolvimento existentes continuam a ser predominantemente baseadas em ambientes locais e mal adaptadas a trabalhos colaborativos ou remotos. Esta dissertação investiga se as tecnologias *web* modernas podem resolver de forma significativa estas limitações e propõe o *PTERO*, um Ambiente de Desenvolvimento Integrado (IDE) remoto *web* para a modelação, implementação e monitorização de aplicações em conformidade com a norma *IEC 61499*.

A arquitetura desenvolvida transfere cargas de trabalho pesadas de orquestração do computador local para um servidor de middleware centralizado. Para resolver o atrito colaborativo inerente a uso tradicional baseado em ficheiros, a plataforma incorpora uma camada de colaboração orientada por Tipos de Dados Replicados sem Conflitos (CRDTs), facilitando a modelação visual simultânea por múltiplos utilizadores.

O sistema foi validado através de testes de desempenho comparativos com o *4DIAC-IDE*, uma ferramenta popular para o desenvolvimento de *IEC 61499*. A avaliação demonstrou que o *PTERO* reduziu substancialmente a sobrecarga de configuração e melhorou as capacidades de colaboração do sistema. Além disso, um estudo de usabilidade confirmou a perceção dos participantes relativamente a estas melhorias. Esta investigação sugere que a migração para um paradigma de engenharia na *web* moderniza a automação industrial distribuída.

Abstract

Cyber-Physical Production Systems (CPPSs) place increasing demands on the software tools used to design and operate distributed industrial control applications. The *IEC 61499* standard provides a structured and event-driven function block model well suited to these environments, yet existing development tools remain predominantly desktop-based and poorly adapted to collaborative or remote workflows. This dissertation investigates whether modern web technologies can meaningfully address these limitations and proposes *PTERO*, a remote web-based Integrated Development Environment (IDE) for the modelling, deployment and monitoring of *IEC 61499*-compliant applications.

The developed architecture shifts heavy orchestration workloads from local workstations to a centralized middleware server. To resolve the collaborative friction inherent to traditional file-based workflows, the platform incorporates a collaboration layer driven by Conflict-free Replicated Data Types (CRDTs), facilitating concurrent multi-user visual modelling.

The system was validated through performance benchmarks against *4DIAC-IDE*, an established tool for *IEC 61499* development. The evaluation demonstrated that *PTERO* substantially reduced configuration overhead and improved the collaboration capabilities of the system. Furthermore, a usability study confirmed the participants' perception of these improvements. Ultimately, this research suggests that migrating to a web-based engineering paradigm successfully modernizes distributed industrial automation.

UN Sustainable Development Goals

The United Nations Sustainable Development Goals (SDGs) provide a global framework for achieving a better and more sustainable future for all. It includes 17 goals[1] to address the world's most pressing challenges, including poverty, inequality, climate change, environmental degradation, peace, and justice.

This dissertation contributes to the development of accessible, collaborative and efficient software tooling for industrial automation systems. By lowering the barriers to developing and deploying *IEC 61499*-compliant distributed control applications, this work supports the broader goals of sustainable industrial development and technological inclusivity. The specific Sustainable Development Goal addressed is:

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

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

SDG	Target	Contribution	Performance Indicators and Metrics
8	8.2	Elevating technological productivity by shifting industrial engineering tasks from fragmented local setups to integrated, cloud-native automated workflows.	Reduction in engineering setup time and configuration overhead during deployment tasks.
9	9.4	Simplifying *IEC 61499* deployment lowers the cost and effort of adopting modern industrial automation practices.	Reduction in development effort compared to existing tools, as measured by workflow comparison and user evaluation.
	9.b	Open, easily accessible tooling fosters industrial innovation in academic and research contexts where commercial tools are inaccessible.	Tool requires no per-user installation and is accessible across all major operating systems.

[1] https://sdgs.un.org/goals

Acknowledgements

I would like to express my gratitude to everyone that contributed to the realization of this dissertation. I give my sincere thanks to:

My supervisor, Rui Pinto, for all confidence put onto me and the help throughout all the work: the valuable insights, the scientific advice and the experience passed on.

Mariana Coimbra for her constant support, encouragement and feedback during this process. Her understanding and motivation were invaluable in the more demanding stages.

I would also like to thank my family and friends for their continuous support and encouragement, as well as everyone who contributed directly or indirectly to this project, whether through feedback, discussions or assistance during testing and evaluation.

Pedro Beirão

"Programming is not a zero-sum game. Teaching something to a fellow programmer doesn't take it away from you. I'm happy to share what I can, because I'm in it for the love of programming."

John Carmack

Contents

List of Figures

List of Tables

List of Listings

List of Acronyms

CI	Continuous Integration
CPPS	Cyber-Physical Production System
CRDT	Conflict-free Replicated Data Type
DSL	Domain-Specific Language
FBD	Function Block Diagram
FEUP	Faculdade de Engenharia da Universidade to Porto
HMI	Human-Machine Interface
ICS	Industrial Control Systems
IDE	Integrated Development Environment
MEEC	Master in Electrical and Computer Engineering
PLC	Programmable Logic Controller
OT	Operational Transformation
RTE	Runtime Environment
SDG	Sustainable Development Goal
SINF	Information Systems
SOA	Service-Oriented Architecture
SUS	System Usability Scale
SYSTEC	Research Center for Systems and Technologies
UI	User Interface

Chapter 1

Introduction

1.1 Context

Modern industrial production increasingly relies on Cyber-Physical Production Systems (CPPSs) [22], which tightly couple computational intelligence with physical manufacturing processes to enable adaptive, data-driven operation. Engineering these systems requires a diverse set of software approaches, among which Function Block Diagram (FBD) [6] has become widely adopted in industrial automation. Rather than expressing control logic as lines of code, FBD represents it as a network of interconnected graphical blocks, each encapsulating a discrete function. This visual paradigm offers tangible advantages: the structure of a program is immediately apparent, individual blocks can be reused across projects and the relationships between system components are explicit rather than implied. For these reasons, FBD is a natural fit for controlling the machinery and processes found in manufacturing environments, often in conjunction with Programmable Logic Controllers (PLCs) [26, 28].

Despite these advantages, FBD introduces tooling demands that text-based languages do not. Constructing and editing a function block network requires specialized software capable of managing graphical composition which is not a concern in a conventional code editor. The overhead this imposes has been documented in both industrial and academic literature, with studies noting that the complexity of graphical FBD workflows compares unfavourably to the relative directness of text-based approaches [8].

In industrial automation, it is common to use the *IEC 61499* standard, which defines an event-driven FBD approach for designing control software [33]. Unlike traditional PLC programming languages, *IEC 61499* emphasizes modularity, reusability and explicit separation of events and data, allowing developers to design complex distributed systems in a scalable and maintainable way. Function blocks define both algorithms and interfaces for inputs, outputs and events making them ideal for CPPS development [10]. This approach is particularly suitable for distributed automation systems, where control logic must respond to events generated by multiple devices or sensors in real time. An example of such a function block interface is shown in Figure 1.1, highlighting the event and data connections that enable flexible interaction between components.

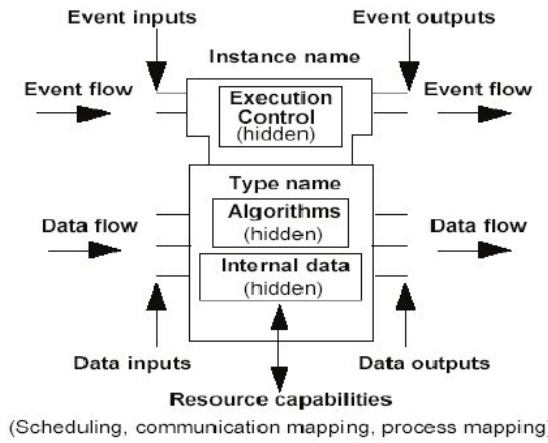


Figure 1.1: *IEC 61499* function block interface [2].

Runtime Environments (RTEs) provide the underlying execution layer for *IEC 61499* applications, responsible for running function blocks on target devices in accordance with the standard's event-driven execution model. In distributed CPPS environments, RTEs are deployed across multiple devices, enabling the execution of decentralized control logic while maintaining coordination between system components.

Conversely, Integrated Development Environments (IDEs) are responsible for the design, configuration, and deployment of applications into CPPS infrastructures. In this role, they act as the primary interface between the engineer and the distributed runtime system, bridging the gap between high-level system modelling and low-level execution. IDEs typically provide a graphical User Interface (UI) as shown in Figure 1.2, with specialized tools for constructing function block networks, configuring device mappings, and managing deployment workflows.

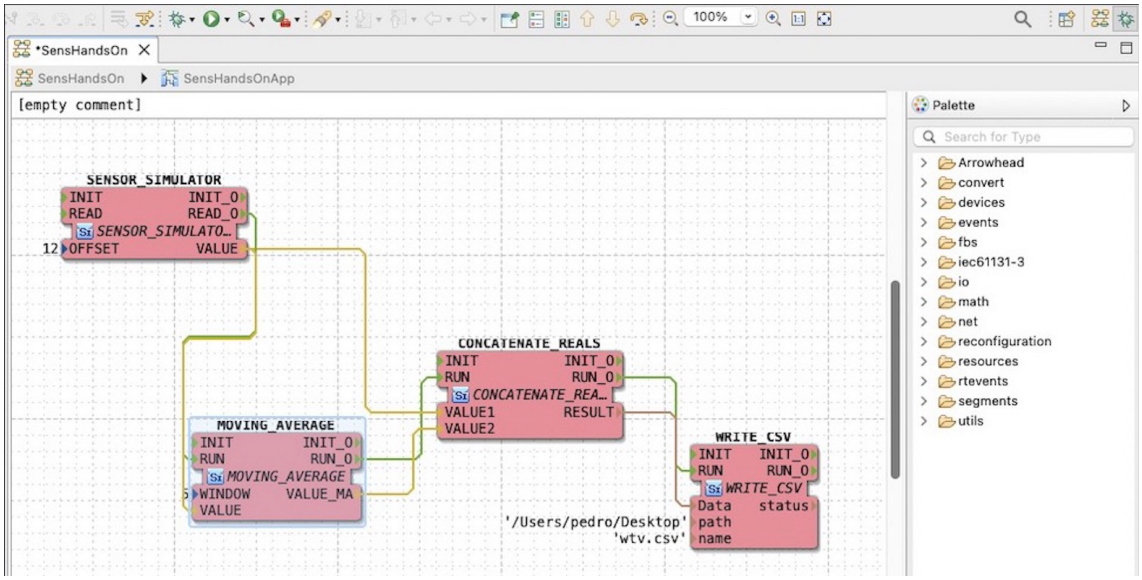


Figure 1.2: 4DIAC-IDE [1].

The relationship and configuration data flow between these two separate operational layers require systematic communication protocols. As illustrated in Figure 1.3, the IDE translates the abstract canvas into deployment payloads transmitted directly to each target device.

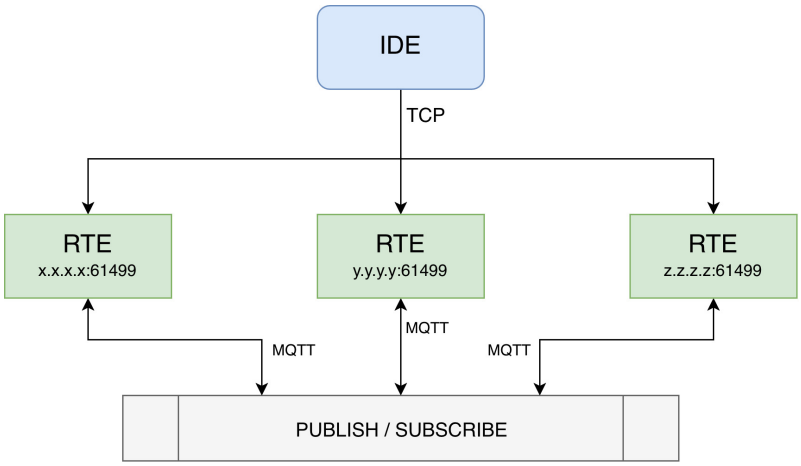


Figure 1.3: Communications between IDE and RTEs.

1.2 Problem

Existing *IEC 61499* development environments lack modern capabilities such as remote access, real-time collaboration and seamless integration with cloud infrastructures. Most are monolithic desktop applications requiring local installation and manual configuration, which hinders efficient development, maintenance and deployment of control applications within distributed CPPSs. Engineers working across dispersed workplaces or with external partners cannot easily share a live view of an application, collaborate concurrently, or access deployed systems remotely without cumbersome workarounds. As a result, current tools fall short in supporting the flexibility and scalability required by modern industrial automation systems.

Besides this, the complexity of these current IDEs rules them out for many simpler use cases where the time wasted on setting up and learning them is not justifiable, limiting the practical accessibility and broader adoption of the *IEC 61499* standard.

1.3 Objectives

This research seeks to investigate how modern web and distribution technologies can be leveraged to enhance the development, deployment and operation of *IEC 61499*-compliant systems. This study is guided by the following hypothesis:

> **Hypothesis**: A remote web-based IDE can improve the modelling, deployment and execution of *IEC 61499* applications.

To further refine the scope, the following research questions are proposed:

RQ1: To what extent can collaborative web technologies improve *IEC 61499* engineering workflows compared with traditional desktop IDEs?

RQ2: What are the performance and usability implications of adopting a web-based collaborative architecture for *IEC 61499* engineering?

1.4 Contributions

This dissertation contributes to the field of *IEC 61499* engineering by investigating the applicability of modern web technologies to the development of distributed automation systems. The main contributions of this work are:

- The design and implementation of Programming Tool for Editing and Runtime Orchestration (*PTERO*), a web-based IDE for the development, deployment and monitoring of *IEC 61499* applications.
- The development of an architecture that integrates modelling, function block management, deployment and monitoring within a single browser environment, reducing the fragmentation commonly found in existing workflows.
- The incorporation of real-time collaborative editing capabilities, enabling multiple users to work simultaneously on the same application.
- An analysis of the benefits and limitations of adopting a web-based collaborative architecture for *IEC 61499* engineering, providing insights for future research and development in this area.

1.5 Dissertation Structure

Besides the Introduction, this dissertation contains five more chapters. The second chapter, Background and Related Work, reviews the current ecosystem, surveying existing *IEC 61499* development environments, collaborative modelling techniques and the evolution of web-based engineering tools, concluding with a comparative analysis that identifies the research gap this work addresses. The third chapter, Proposed Solution, details the full system proposal, from the server architecture to the collaboration and deployment models, and in turn, the fourth chapter, Implementation, describes the implementation of said proposal. It documents how the solution can be built for real world use, and the decisions associated. The next chapter, Validation, examines the results derived from the developed prototype. This chapter contains sections that analyse each specific evaluation performed: the functional validation, the performance evaluation, the collaboration evaluation and the usability evaluation. Finally, the last chapter, Conclusion, overviews the document's topics and presents the main conclusions. The document ends with the idealisation of possible future work unlocked by this project, aiming to guide future investigation.

Chapter 2

Background and Related Work

This chapter reviews prior work related to *IEC 61499*-based development and the used IDEs. The goal is to better understand the landscape, which will then allow the critical review of the state of the art. By providing a comprehensive analysis of the current landscape of development environments for CPPSs and the *IEC 61499* standard, we are able to categorize and critically examine the existing solutions, while looking for possible gaps in the ecosystem.

2.1 Search and Selection

A targeted literature search was conducted across the digital libraries *ScienceDirect*, *IEEE Xplore*, *Scopus* and *ACM Digital Library*, complemented by the research aggregator *Semantic Scholar*. The search was restricted to publications from 2008 onward, in order to capture the period following the consolidation of the *IEC 61499* standard, and to peer-reviewed journal articles and conference papers written in English. The following core terms were identified as the most promising:

- *IEC 61499*
- FBD
- IDE
- CPPS

Related terms that were discovered mid-search were also included in some queries. The objective of doing this was to get a better personal understanding of the scene, as well as to find interesting references that could be included in this dissertation to back up the written claims. Some of the terms used were:

- 4diac
- remote

- collaboration
- real-time

For each database, queries were built by combining the identified terms using boolean operators. The base query, adapted to the syntax of each platform, took the form:

```
("IEC 61499" OR "function block diagram") AND ("IDE" OR "development
environment")
```

For example, in *Scopus* this was expressed using the `TITLE-ABS-KEY` field as:

```
TITLE-ABS-KEY(("IEC 61499" OR "function block diagram") AND ("IDE"
OR "development environment")) AND PUBYEAR > 2007
```

In the other platforms the same term combinations were applied to the *Title, Abstract, Author Keywords* fields, using each respective advanced search syntax.

This initial search returned a total of 43 unique records across the databases. Papers were excluded if they were out of scope or were inaccessible in full text. This screening stage reduced the set 31 papers, used for detailed analysis and citation in this dissertation. Next, all retained references were imported into *Zotero* for management and citation formatting. The reference lists of the selected papers were manually inspected to identify additional relevant works not captured by the initial keyword search, this process surfaced 8 further papers that were incorporated into the review.

Articles proposing specifically new *IEC 61499* development environments were prioritised for inclusion, as their own related work sections typically contained an independent gap analysis, allowing the present review to be cross-checked against the prevailing assessment of the scientific community.

2.2 Distributed Function Block Engineering

Modern manufacturing and industrial automation are increasingly characterized by the transition from monolithic centralized control architectures towards distributed modular systems. The industry is moving towards a paradigm in which individual parts possess intelligence and communicate with one another. This shift is driven by the demands of Industry 4.0, which calls for flexibility and interoperability as key properties of the production infrastructure [9]. Monostori [22] provides a comprehensive analysis of CPPS, examining the theoretical foundations and practical implications of tightly coupling computational intelligence with physical manufacturing processes. The author explores how cyber-physical integration enables real-time monitoring, adaptive control and autonomous decision-making in production systems, framing CPPS as a cornerstone of the fourth industrial revolution. This work establishes the concepts, wording and architectural principles that help understand much of the research in this area, and it serves as a reference for understanding why the flexible distributed control is necessary to realize the vision of CPPS.

The *IEC 61499* standard was developed precisely to address these requirements at the software level. This architecture defines a domain-specific modelling language for developing distributed Industrial Control Systems (ICS). It does so by extending and superseding the older *IEC 61131-3* standard, which was designed for centralized PLC-based control, providing next-generation systems with an architecture that enables distributed control [5]. The architecture encourages flexible reuse of components as well as reliable data exchange.

In FBD, function blocks are interconnected into networks that form an application, which can then be distributed across multiple devices [18], a property that makes FBD a natural fit for CPPS deployments where hardware must collaborate to execute a unified control program. Torres, Gonçalves, and Pinto [36] conducted a literature review examining how *IEC 61499* can be leveraged specifically for CPPS development, mentioning possible enhancements with the use of a cloud-based Service-Oriented Architecture (SOA) approach. Their paper examines the current state of adoption of the standard within industrial automation research, identifying recurring architectural patterns in CPPS implementations.

Conversely, Pinto *et al.* [28] presented an industrial case study showing that function block oriented approaches improve system modularity and reduce development effort compared to traditional procedural programming. Their study documents the migration of a production cell from a sequential control architecture to an *IEC 61499* function block model using the Dynamic INtelligent Architecture for Software and MOdular REconfiguration (*DINASORE*) RTE, using integration time as the main validation metric and demonstrating improved reconfigurability when production requirements changed. The results show that the encapsulation of behaviour and communication interfaces within function blocks leads to more maintainable and extensible software.

2.3 IEC 61499 Development Environments

Currently, the development ecosystem for *IEC 61499* is composed almost exclusively of desktop-based IDEs. These tools are typically monolithic applications installed locally on engineering workstations, requiring each member of a development team to independently install, configure and maintain the software on their own machine. The tools themselves tend to be resource intensive, which creates demands for the hardware, making them less accessible to users working on modest, shared or virtual machines. There is also the issue of platform dependency, as a niche and focused field, some IDEs are not cross-platform, forcing users to use virtual machines to be able to use them, further intensifying performance issues.

2.3.1 Open-Source

The most prominent open source implementation is the *4DIAC-IDE* (based on the *Eclipse* framework) which supports a variety of RTEs such as *4DIAC FORTE* and *DINASORE*. This IDE is very feature rich at the cost of increased complexity, resulting in a steep learning curve, which makes it difficult to use [39]. The *Eclipse* foundation on which it is built is itself a heavyweight platform and while this grants access to a broad plugin ecosystem, it also means that the IDE

inherits the framework's architectural complexity. Navigating the interface to accomplish basic tasks such as creating a new function block type, designing the flow and deploying to an RTE can require substantial initial investment before productive work can begin.

Zoitl, Strasser, and Ebenhofer [41] demonstrated the use of *4DIAC-IDE* to develop modular and reusable *IEC 61499* applications, highlighting the benefits of flexibility and adaptability in industrial settings. The paper provides a practical perspective on the strengths and limitations of this IDE.

An alternative is *FBME*, which is a much simpler modelling environment based on the *JetBrains MPS* language workbench. This IDE implements *IEC 61499* concepts through a set of integrated Domain-Specific Languages (DSLs). The main contribution of this work is the research on applying model-driven development principles to the architecture of the IDE itself [33], resulting in a very modular and extensible product.

2.3.2 Commercial

The commercial side is occupied by *EcoStruxure Automation Expert*, *ISaGRAF Workbench* [38], tools that have their own built-in proprietary RTE. This tight coupling between the engineering tool and the runtime gives commercial vendors significant control over the end development experience, which typically results in polished, production-ready tooling with strong vendor support. However, it also introduces considerable lock-in: projects developed in one commercial environment are not easily portable to a different runtime or IDE and the proprietary nature of the underlying RTE limits the ability to inspect, extend, or adapt behaviour.

Commercial tools are also constrained by their licensing models. Per seat licensing is common, meaning that the cost of the toolchain scales directly with the size of the team and can not be viable for smaller organizations, academic research groups or early stage projects. This creates a tiered ecosystem in which well resourced industrial organizations have access to mature tooling, while smaller teams are restricted to the open source alternatives, each of which carries its own set of limitations as described above.

2.3.3 Local-first Development

All the tools mentioned previously share a fundamental design decision: they play around a local-first development model, in which the project state lives on a single workstation. While this model has historically been the norm in industrial automation tooling, it introduces a set of limitations that become problematic as CPPS projects grow in scale and involve more distributed engineering teams.

The most immediate consequence is the absence of real-time collaboration. When two engineers need to work on the same *IEC 61499* application simultaneously, they must coordinate manually, typically by dividing the project into separate files or modules and merging their changes afterwards. These tools also require the IDE to maintain awareness of the deployment target's

state. Without a shared backend, each engineer must independently configure their local environment to point to the correct runtime endpoints and therefore there is no single source of truth for the current deployment state of the system.

2.4 Collaborative Modelling

As projects grow in scale and involve distributed engineering teams, the question of how to coordinate concurrent modifications to a shared application model becomes central [13]. Version control through *Git* is inadequate in this domain as it operates on a line based difference algorithm designed for plain text source code. In the general case, merging structured data files through line-based operations is inherently unreliable, the result may be syntactically valid yet semantically broken, as the merge has no awareness of the relationships between elements. *IEC 61499* applications are stored as *XML* documents whose semantic integrity depends on relationships between elements. A valid merge operation can still result in a broken application graph.

Approaches to real-time concurrent editing, Operational Transformation (OT) and Conflict-free Replicated Data Type (CRDT) offer a fundamentally different model of collaboration from the workflow of version control systems. Rather than requiring engineers to work in isolation and later reconcile divergent states, these techniques allow all participants to apply changes optimistically to their local replica, with the system guaranteeing that every replica converges to the same state. This shift ensures that inconsistencies are resolved continuously and automatically, rather than discovered only at integration time, enabling a concurrent engineering experience over shared structured artefacts such as *IEC 61499* application graphs.

2.5 Evolution of Web-Based Tools

The software engineering domain has seen a massive shift from desktop IDEs to cloud based environments. This trend is now influencing the domain of ICS and while recent studies have proposed a SOA to integrate *IEC 61499* with cloud layers [36], the engineering tools themselves remain still desktop bound.

In general software development, tools like *Visual Studio Code* have started offering web based counterparts to their desktop applications [37] in search of a more flexible user experience. Running entirely within the browser, these environments allow engineers to open, edit and navigate codebases from any device without installation. They integrate naturally with cloud hosted codebases and Continuous Integration (CI) pipelines.

Web based editors are also often easier to extend with real-time collaboration because they operate within a network architecture, leverage standardized browser communication technologies such as WebSockets and naturally integrate with centralized synchronization services. These characteristics reduce many of the distribution and interoperability challenges that desktop applications must address separately [31].

The move towards web based tooling is not limited to software development environments. Across a wide range of professional domains, browser-native applications have gradually replaced traditional desktop software. Collaborative document editors such as *Google Docs* allow multiple users to edit the same document simultaneously, with changes synchronized in real time. Similarly, design and prototyping platforms such as *Figma* have demonstrated that even complex graphical workflows can be delivered entirely through the browser while supporting collaboration and centralized asset management. Drawing tools like *Excalidraw* and *draw.io* further illustrate how browser based applications can provide sophisticated modelling and visualization capabilities without requiring local installation, while providing live collaboration features that their desktop predecessors did not have.

A common characteristic of these platforms is that they treat collaboration as an architectural design rather than an add-on. By maintaining a shared representation of the artefact on a central service, users can concurrently make changes with minimal configuration. This contrasts with many traditional desktop applications, where collaboration often relies on exchanging files with separate version control and synchronization mechanisms. As a result, web based tools have created a new expectation on collaboration.

Some studies explore remote web based execution and editing of FBD. Rohat and Popescu [30] proposed an early approach to remote monitoring and interaction with *IEC 61499* applications through a web interface, identifying the practical benefits of browser access in distributed industrial settings. While the work predates many modern web technologies, it established the motivation for moving development and monitoring capabilities to the browser. More recently, Lyu *et al.* [21] revisited this direction by proposing fully remote cloud based platform for *IEC 61499* development, embedding *EcoStruxure AE* in a frontend web interface. This way they were able to move all the processing into a centralized cloud platform.

More generally, Sorokin and Vyatkin [34] suggests a new architecture for the development of web-based DSL tools, grazing on the field of *IEC 61499* engineering by using *FBME* as an example of how it can be migrated into a distributed collaborative web environment. While this paper does not provide the groundwork and validation necessary for the development of specifically an *IEC 61499* tool, it does provide good general insights of how its architecture could be designed.

2.6 Comparative Analysis and Research Gap

Having surveyed the existing landscape of *IEC 61499* development environments, both established tools and recent proposals, it is possible to draw a structured comparison across the dimensions most relevant to modern CPPS engineering. Table 2.1 summarizes the key characteristics of each solution with respect to platform, remote deployment capability and support for real-time collaboration.

Current *IEC 61499* tools face significant limitations that hinder modern CPPS development. These gaps paint a picture of an ecosystem that has matured in terms of standards compliance, but

Table 2.1: Comparison of IEC 61499 development environments and proposals.

Tool	Type	Platform	Remote deployment	Real-time collaboration	Comments
4DIAC-IDE [41]	Open source	Desktop	×	×	Steep learning curve
FBME [33]	Open source	Desktop	×	×	Very modular
EcoStruxure AE	Commercial	Desktop	×	×	Proprietary ecosystem
ISaGRAF Workbench [38]	Commercial	Desktop	×	×	Licensing model
Rohat and Popescu [30] (2014)	Proposal	Web (monitoring only)	✓	×	No remote modelling capabilities
Lyu *et al.* [21] (2024)	Proposal	Web (embedded desktop IDE)	✓	×	Does not take advantage of web technologies for the modelling tool
Sorokin and Vyatkin [34] (2023)	Proposal	Web	✓	✓	Designs a general architecture for creating web DSL tooling
PTERO	New Proposal	Web	✓	✓	-

has lagged considerably in adapting to the workflows and expectations of contemporary engineering teams. The following points describe the most significant gaps:

- **Platform Dependency:** Existing solutions are monolithic desktop applications, lacking the accessibility and portability that we have come to expect. Engineers are required to install and maintain native software on every machine from which they intend to work, creating friction at the start of projects and compounding over time as the team grows or its members change.
- **Absence of Real-Time Collaboration:** None of the existing tools provide support for real-time editing. Teams must rely on external version control systems and manual coordination to manage shared development, introducing overhead and increasing the risk of integration errors that would be avoided by a collaboration aware environment.
- **Lack of a Centralized System State:** Since existing tools are local-first, there is no single representation of a system's current state. This makes it difficult to maintain a consistent view of the system across a distributed team to integrate the development environment into automated deployment and monitoring pipelines.
- **Limited Accessibility:** Engineers cannot access a project without access to a fully configured workstation. This is a practical limitation in industrial settings, where on-site diagnosis of a running application may be necessary from a device in which no local tooling has been installed, and the project is not set up.

Taken together, these gaps suggest that the ecosystem is well positioned to benefit from a lightweight, browser-accessible development environment that prioritizes collaboration, low setup overhead and integration with modern deployment workflows, without sacrificing the domain-specific features that *IEC 61499* engineers depend on.

Regarding Sorokin and Vyatkin [34]'s proposal, the closest current solution, it provides valuable architectural groundwork for web-based DSL tooling in general, but their work is deliberately

generic, it is not built around the specific semantics, data model or deployment process of *IEC 61499*, and it offers no working implementation or empirical validation. Migrating *FBME* into a distributed web environment, as suggested in the paper, is presented only as an illustrative example rather than as a concrete, tested system. Consequently, the gaps here identified are still justified, as none are properly addressed by this previous paper to the degree needed.

Chapter 3

Proposed Solution

To address the limitations identified in existing *IEC 61499* development environments, this dissertation proposes the design of *PTERO*, a web-based IDE for modelling *IEC 61499*-compliant applications. The goal is to bridge the gap between modern collaborative software engineering practices and industrial automation tooling used in CPPSs.

Existing tools are predominantly desktop based and poorly suited for distributed development. In contrast, the proposed solution adopts a centralized web-based architecture that enables platform independence, simplified setup, simplified deployment and real-time collaboration.

The main reason to suggest using web technologies is to solve the issues that affect any local-first development environment. By having the IDE be hosted by a centralized server, the following key aspects are achieved:

- **Ease of setup**. Instead of each team member having to install a resource heavy program and set it up with the team's code and configurations, a single server is set up and the team members simply have to connect to it via their browser.
- **Ease of deployment**. Only the server needs to be aware of all the target RTEs and have connection to them. When a team member wants to deploy, the request will be routed through the server, acting as the middleware, removing the need for each development machine to have connection with the target RTEs.
- **Improved collaboration**. A real-time collaboration aware system that keeps track of atomic changes, instead of blocks of text, reducing the integration errors associated with version control systems on structured data.

This research follows a design-oriented research approach. The core of the research consists of the design and implementation of a functional prototype of the solution. Rather than aiming for statistical generalization, the study focuses on demonstrating feasibility and identifying qualitative improvements over existing environments.

3.1 System Architecture

The proposed system follows a centralized server architecture designed to support distributed CPPS development. Instead of the IDEs having direct communication with the RTEs, the server acts as an intermediary, meaning all requests go through it and removing the need for direct device-level configuration on individual machines. Besides serving the frontend to the users, it also stores all data in a centralized database. The development pipeline therefore consists of the following three core components, organized as displayed in Figure 3.1:

- **Frontend IDE**: A web interface for setting up and modelling the *IEC 61499* applications.
- **Server Middleware**: A centralized backend responsible for state management, collaboration and communication.
- **RTEs**: External execution environments where the applications are deployed.

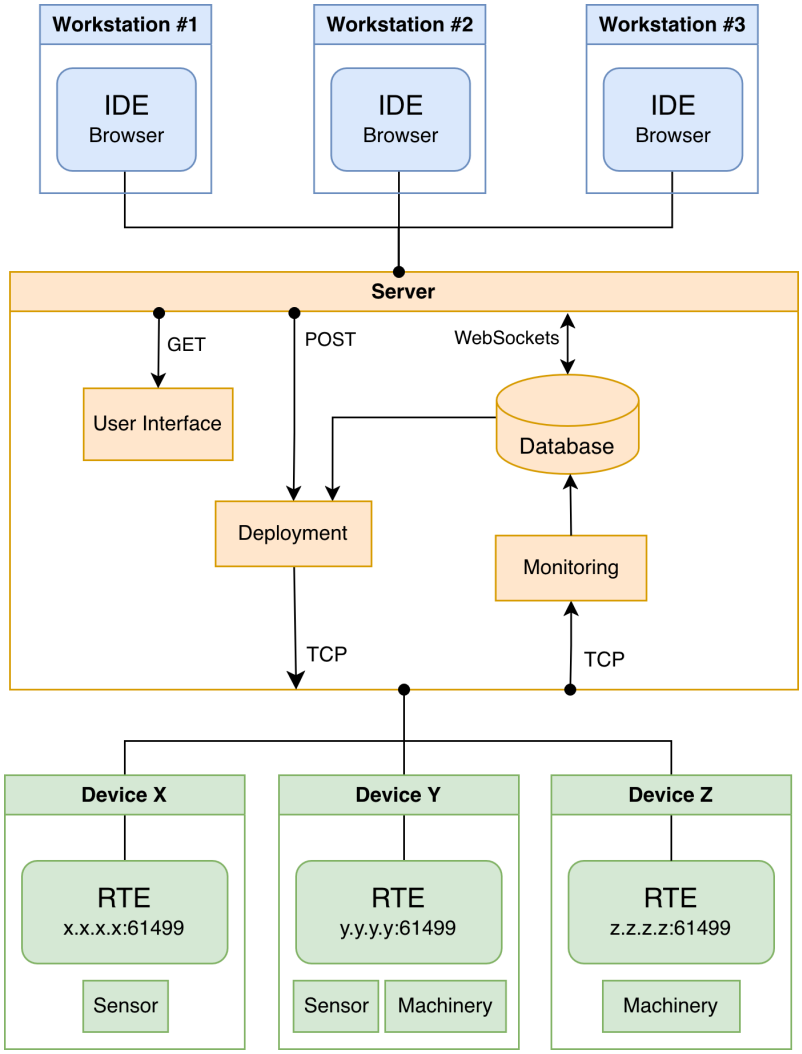


Figure 3.1: Communication through the server.

Being the middleware, there are 4 different types of network communications that the server will need to handle, each with its own requirements:

- **Serve webpage**: Communicates via *GET* requests to deliver the *HTML*, *CSS* and *JS* files that make up the IDE interface.
- **Real-Time Collaboration**: Frequent atomic updates whenever changes are made.
- **Requested deployment**: A team member sends a *POST* request, that tells the server to deploy.
- **Deployment sockets**: Communicates via *TCP* with each RTE to send commands and monitor.

3.2 Real-Time Collaboration Model

A key requirement of the proposed system is support for concurrent editing of structured *IEC 61499* programs. Real-time collaborative systems shift conflict management from the merge phase to the editing phase itself, all participants may operate on the shared artefact simultaneously, with the system responsible for converging divergent states automatically. This stands in contrast to the conventional engineering workflow, where each participant maintains a local copy of the project and conflicts are only discovered when changes are reconciled, typically through a version control system. For graph-based artefacts such as *IEC 61499* applications, this delayed-conflict model is particularly disruptive: two engineers independently rewiring the same function block network may produce textually irreconcilable diffs even when their intentions do not actually overlap, simply because the underlying file format encodes structural relationships as serialized text.

To address this, the system adopts a CRDT approach, enabling multiple users to edit shared application models simultaneously. Changes are propagated in real time and merged deterministically without conflicts. Each participant's local replica can therefore be updated optimistically and immediately, with network propagation handled asynchronously in the background, which is essential for maintaining a responsive editing experience even under variable network latency between client and server.

Compared to OT approaches, CRDTs provide a more suitable foundation for non-linear structured data. Where in OT a semantic state change needs to be expressed as a textual change at a specific offset, in CRDT it keeps the semantic meaning [14]. This distinction matters significantly for a graph-based modelling language such as *IEC 61499*. OT was originally designed for linear documents, where edits can naturally be expressed as insertions and deletions at character offsets. A CRDT-based model instead handles operations that are defined directly on the data structures that make up the graph.

While other IDEs store the graph based structured data in conflict prone files in formats such as *XML*, here it should instead be kept as individual fields in a tree based structure. This structure, just like *XML* allows, is instrumental to keeping a representation of the graph; this is why other

tools, like *4DIAC-IDE*, choose to use this format. To combat merge conflicts, a change in a single element should immediately prompt the CRDT to only look into that specific field and not the document as a whole. This field-level granularity ensures that the scope of any potential conflict is reduced to the smallest meaningful unit of change

The graph's state is made up of nodes, also referred to as function block instances, which have a specific function block type and are connected with each other via links. These links are hooked onto certain input and output slots and each node will be ran in the specific RTE assigned. The class diagram in Figure 3.2 illustrates how the fields can be organized.

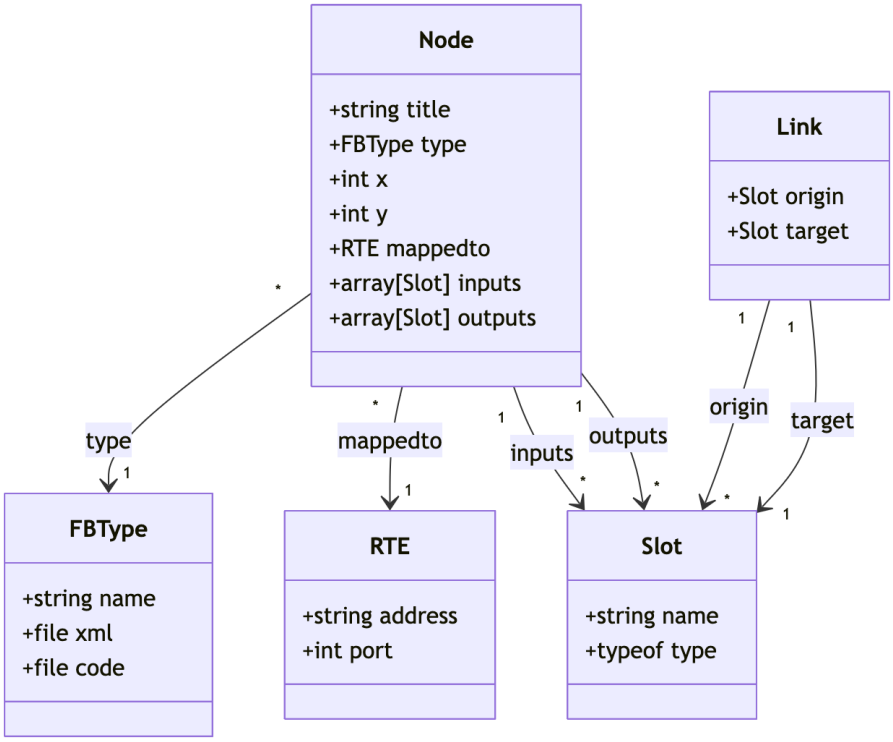


Figure 3.2: Class diagram of the graph's state.

3.3 Deployment Model

Deployment in the proposed system is based on a centralized orchestration model. Rather than requiring each engineer to manually coordinate communication with target devices, the server acts as the single point of control, translating the graphical application model into a sequence of commands and executing them against the configured RTEs. This abstraction ensures that deployment behaviour is consistent and reproducible regardless of which team member initiates it or from which machine. The server consumes the CRDT state directly, guaranteeing that whatever is deployed reflects a single, consistent snapshot of the application as agreed upon by all participants. The deployment process is divided into the following three sequential stages:

1. **Synchronization:** Before any execution commands are issued, the function block type definitions must be present on each target device. The server transfers the relevant files to the appropriate RTE instances, ensuring that every device has access to the types it will be asked to instantiate. This stage exists because an RTE cannot instantiate a function block whose type it does not have access to, it must therefore be transferred ahead of any instantiation command.

2. **Execution:** Once the function block files are in place, the server establishes a connection with each RTE and issues the necessary commands to create function block instances, write initial input values, establish event and data connections and finally start execution. These commands are issued in a fixed order to respect the dependencies inherent in the application model, for example an instance cannot be connected until both its source and destination instances are created. The resulting runtime configuration is fully reproducible. Redeploying the same application model results in an equivalent set of commands sent.

3. **Monitoring:** After the application has started, the server begins polling the RTEs for their output values, making them available to all connected clients through the IDE interface in real time. This closes the loop between the deployed application and the editing environment.

Chapter 4

Implementation

To be able to prove the proposed solution is viable and will successfully address the gap identified, a prototype needs to be developed and afterwards validated. This chapter describes the technical implementation of the proposed system, detailing the server architecture, frontend implementation, collaboration system and deployment mechanism used to realize the conceptual design presented in Chapter 3.

4.1 Runtime Environment

Out of all RTEs compatible with *IEC 61499*, this prototype will focus on supporting *DINASORE* [27]. It is much less complex than others, making it ideal for a proof of concept IDE. As an open-source project, *DINASORE* can be freely inspected and studied, which is particularly valuable in a research and prototyping context where flexibility is prioritized over real-time performance guarantees. This ease of study rules out the use of proprietary RTEs like *EcoStruxure Automation Expert's dPAC* and *ISaGRAF Runtime*.

Where the open-source RTE *4DIAC FORTE* provides a portable C++ execution environment for low-level industrial deployments, *DINASORE* occupies the same architectural role while targeting higher-level workloads that benefit from the *Python* ecosystem. The two runtimes share the same management protocol, meaning that concepts developed against one transfer directly to the other. *4DIAC FORTE* itself was considered as a target RTE but ruled out as its codebase is substantially larger and more complex, making it more complex to study and target.

Figure 4.1 represents the architecture of this *DINASORE* where *PTERO* aims to replace the *4DIAC Interface*.

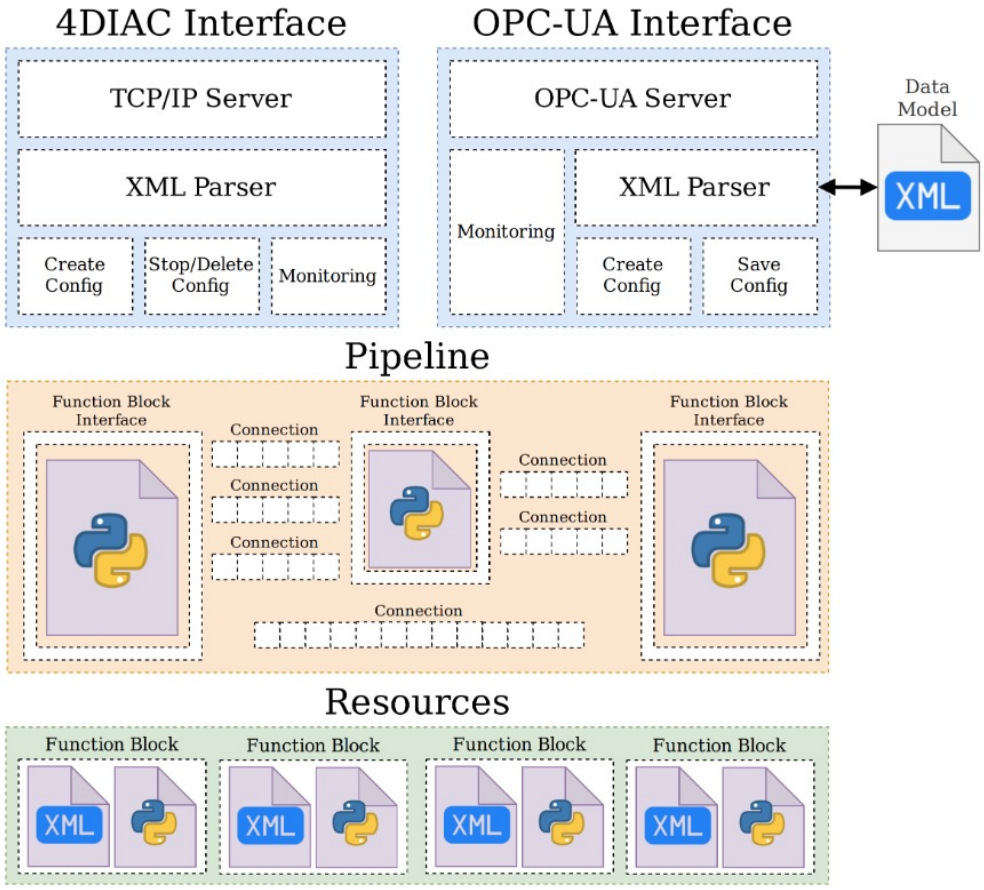


Figure 4.1: *DINASORE*'s architecture [27].

4.2 Server

The design begins by setting up a centralized backend server tasked with serving the IDE interface as a web application, managing data persistence and communicating with the distributed RTEs. Taking into account the features that are planned, *JavaScript* running on *Node.js* was chosen as the underlying environment to run the server on, due to the extensive library ecosystem, leveraging an asynchronous, event-driven and non-blocking I/O model [35].

To address data management, the server adopts a *Document-Oriented Database* system. Because *IEC 61499* configurations, such as function blocks and event/data connections, are inherently structured as hierarchical graphs, a database that stores data as key values pairs represents the optimal architectural fit. In fact, local-first IDEs for the standard usually store the programs in an *XML* format for this exact reason.

Architecturally, the server functions as a multiprotocol hub: it handles traditional HTTP traffic and maintains WebSocket channels for bidirectional communication with active RTEs. For real-time collaboration, a library will be required to run in both the server and the frontend. Since *JavaScript* runs natively in the browser, it further reinforces that it should also be used in the server,

creating a homogeneous single language stack that simplifies data serialization across network boundaries. The library choice will be explained in depth in Section 4.4, proving the previous train of thought.

4.3 Frontend

The frontend constitutes the client-side architectural layer of the system, acting as the primary Human-Machine Interface (HMI) through which users interact with the underlying application logic. In modern web architectures, the frontend is responsible for rendering the UI, capturing user inputs, executing localized presentation logic and managing real-time state synchronization with the backend server. For this specific application, the frontend serves as the interactive environment used to create, visualize and modify FBD programs.

Each following subsection will first explain and describe what an IDE should implement and follow by showing how the prototype solved the proposed requirements.

4.3.1 Configuration View

In the interest of distributed CPPSs, *IEC 61499* allows function blocks to be mapped to different *DINASOREs*, so a clean way to define the resources that are available is needed. Both the *DINASORE* port and the *SSH* port need to be defined for each since deployment communication and synchronization will use each of them, respectively.

Accordingly, each configuration profile maintains these essential properties:

- **Name**: Human friendly name to quickly identify what device this is
- **Address**: The address of the device
- **DINASORE port**: The port where *DINASORE* is running
- **Colour**: The colour associated with this *DINASORE* instance, providing a visual distinction in the *Graph View* between function blocks mapped to different instances.
- **SSH port**: The open *SSH* port of the device.
- **SSH user** and **SSH password**: The *SSH* login credentials.
- **SSH path to DINASORE**: Path in the *SSH* filesystem pointing to where *DINASORE* is located.

With the existence of the middleware, this solution removes the requirement for each team member to configure the available *DINASORE* instances in their own IDE and instead keeps a shared configuration in the server database. This was implemented in the prototype as can be seen in Figure 4.2.

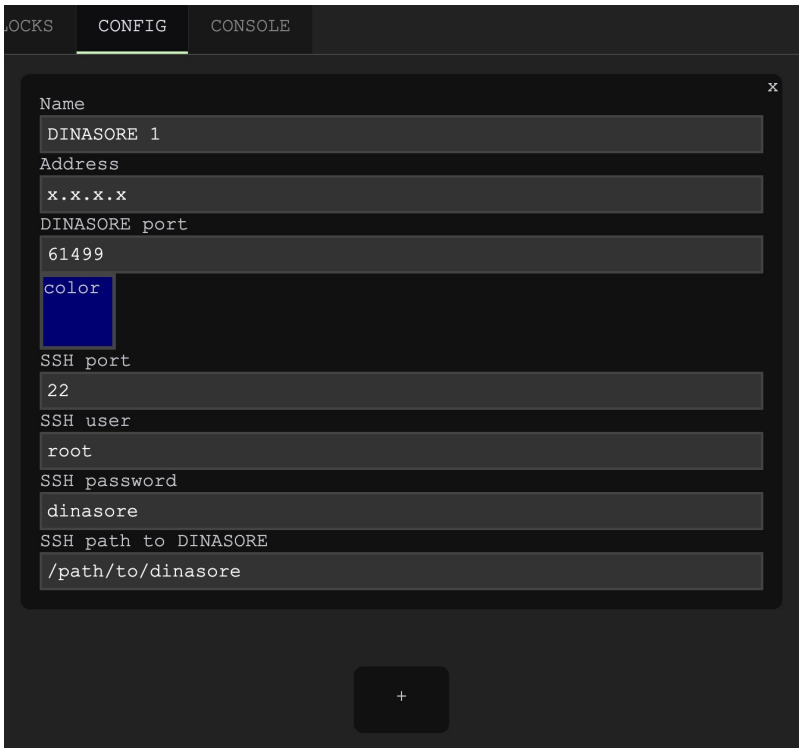


Figure 4.2: Configuring a *DINASORE* in the Configuration View.

4.3.2 Function Blocks Editor View

This is where individual function blocks are defined. It allows the team to specify the interface of each function block, its input and output data ports and events, as well as program the internal behaviour. This separation of interface and implementation follows the *IEC 61499* model of encapsulation, where a function block exposes a well-defined interface while keeping its internal logic self-contained.

Like in the other RTEs, in *DINASORE* the interface of the function block is defined in an *XML* format file with a *.fbt* extension, which results in a function block that can be displayed graphically like seen in Figure 4.3. The internal behaviour of the function blocks in *DINASORE* is programmed in *Python*.

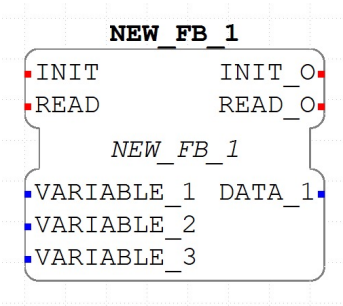


Figure 4.3: Example function block defined via *XML*.

For the IDEs to be able to detect and use the function block, both the *.fbt* file and the *.py* file need to be placed in a specific folder, right next to each other. In this prototype this is handled automatically in the server database, removing the need to manually move files like it is needed in *4DIAC-IDE*.

As seen in Figure 4.4, this view contains a split view text editor: a side for the *.fbt* file and the other for the *.py* file. It would be possible to create a visual editor for designing the interface, instead of having to write *XML* by hand, but the added complexity would not be justified at this prototyping stage. The *XML* format is sufficiently structured that a text editor provides an adequate experience and a dedicated visual interface remains a natural direction for future development.

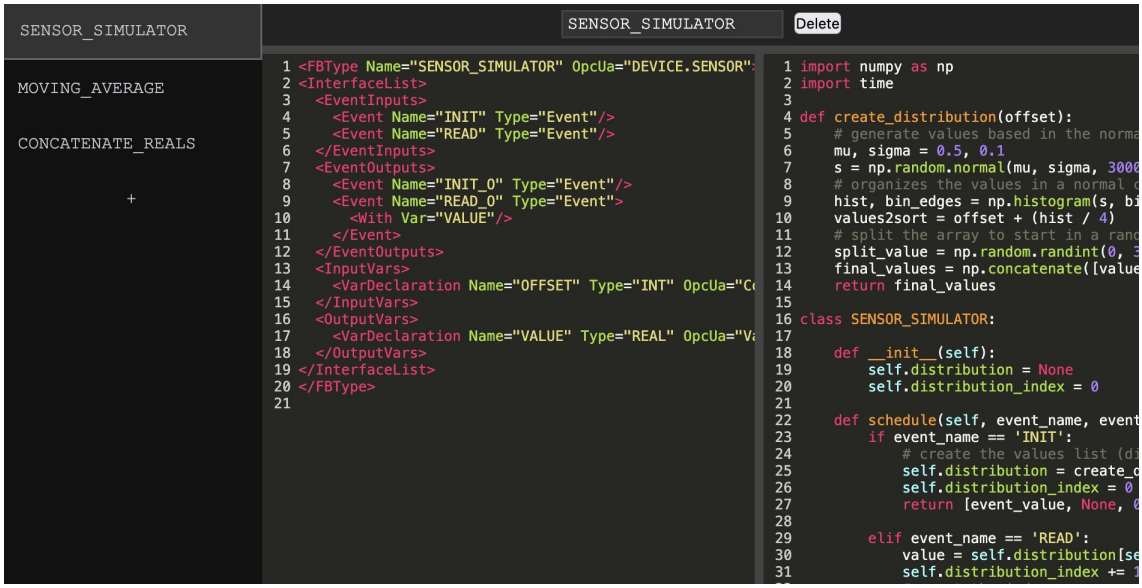


Figure 4.4: Editing function block files in the Function Blocks Editor View.

4.3.3 Graph View

This view is the primary workspace for composing the application. Function blocks are placed on the canvas and connected together by drawing connections between their data and event ports, forming the application network. Event ports are commonly colour coded red, while data ports are colour coded blue.

A function block will have some input and output data ports, each with its own associated data type. For example, an *INT* input port will not accept data from a *WORD* output port and therefore should not be able to connect to it. It is also possible to set a default value for an input if no output is connected to it, there is an editable string field for this which gets converted to the expected data type by the RTE. The data types supported by *IEC 61499* are derived from and defined by *IEC 61131-3* as listed in Table 4.1.

Type	Description	Size
BOOL	Boolean	1-bit
SINT	Short integer	8-bit
INT	Integer	16-bit
DINT	Double integer	32-bit
LINT	Long integer	64-bit
REAL	Floating point	32-bit
LREAL	Long floating point	64-bit
STRING	ASCII string	Variable
WSTRING	Unicode string	Variable
TIME	Time interval	
DATE	Date	

Table 4.1: IEC 61131-3 Data Types [12].

Each function block can also be mapped to a *DINASORE* instance, with the colour coding defined in the *Configuration View* providing an immediate visual indication of how the application is distributed across the available devices.

This system was achieved in the prototype with the use of the *LiteGraph.js* library [19], a graph node editor for the browser. A lot of other libraries were considered, but this one was chosen due to the requirements of the proposed solution. Let us discuss the alternatives that were considered to understand why this choice is important:

- **Rete.js** [29]: A very flexible and complete editor framework, but the added complexity makes it harder to integrate the desired features. It is also a more general purpose editor, meaning the multi-input and multi-output function blocks with specific data types are not as well-supported.

- **xyflow** [25]: A highly customizable framework composed of React Flow and Svelte Flow. While it provides excellent visual customization and integration with frontend frameworks, it suffers from the same problems as *Rete.js*.

- **Drawflow** [32]: A lightweight and simplistic library with a focus on data flow execution. Although simple to integrate, it lacks many features required for designing *IEC 61499* function blocks, such as robust type handling and multi-input and multi-output.

- **Baklava.js** [4]: With a clean architecture and flexibility, this would be a good alternative, but since it specifically targets *Vue* and requires a more intricate setup, it was not the most appropriate choice for a prototype.

- **GoJS** [11]: Very powerful commercial alternative. It offers advanced functionality and flexibility, but its licensing model make it less suitable for lightweight research prototypes and open development environments.

- **Node-RED** [20]: A flow based programming environment. While conceptually similar, its architecture is more focused on runtime execution than on being a visual editor of structured data.

Among these alternatives, *LiteGraph.js* presented the best balance between simplicity, flexibility and ease of integration. Its built in support for typed inputs and outputs, lightweight architecture, and direct focus on visual data flow programming made it especially suitable for the requirements of the proposed solution. Even so, it was necessary to implement custom behaviour on top of some functions of *LiteGraph.js* to achieve the desired behaviour. The choice of the graph library is very important, since the requirements of *IEC 61499* differ significantly from those of most conventional graph editors, as such, special care must be taken to ensure that these requirements are properly supported. The described implementation in the prototype is displayed in Figure 4.5.

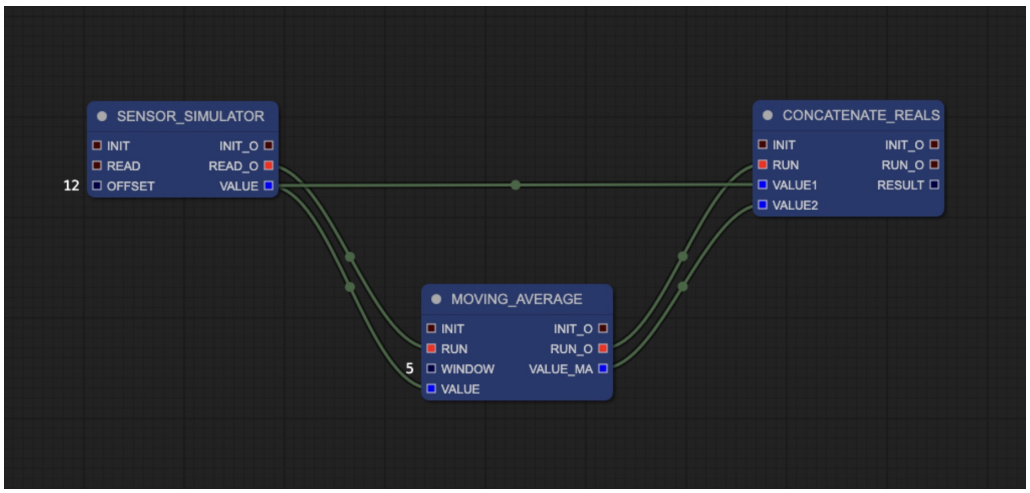


Figure 4.5: Configuring the data flow in Graph Editor View.

4.3.4 Console View

This view, as shown in Figure 4.6, serves as the primary logging interface within the frontend architecture. It displays the outputs generated during the deployment process. The explicit detailed phases of these logs are discussed further in Section 4.5.

```
python3 ./sync/synchronize.py
Starting copying process
Copying process completed
Synchronized 1 DINASOREs
<Request Action="QUERY" ID="0"><FB Name="*" Type="*"/></Request>
<Response ID="0" />
<Request Action="CREATE" ID="1"><FB Name="EMB_RES" Type="EMB_RES"/></Request>
<Response ID="1" />
<Request Action="CREATE" ID="2"><FB Name="SENSOR_SIMULATOR" Type="SENSOR_SIMULATOR"/></Request>
<Response ID="2" />
<Request Action="WRITE" ID="3"><Connection Destination="SENSOR_SIMULATOR.OFFSET" Source="12"/></Request>
<Response ID="3" />
<Request Action="CREATE" ID="4"><FB Name="MOVING_AVERAGE" Type="MOVING_AVERAGE"/></Request>
<Response ID="4" />
<Request Action="WRITE" ID="5"><Connection Destination="MOVING_AVERAGE.WINDOW" Source="5"/></Request>
<Response ID="5" />
<Request Action="CREATE" ID="6"><FB Name="CONCATENATE_REALS" Type="CONCATENATE_REALS"/></Request>
<Response ID="6" />
<Request Action="CREATE" ID="7"><Connection Destination="MOVING_AVERAGE.VALUE" Source="SENSOR_SIMULATOR.VALUE"/></Request>
<Response ID="7" />
<Request Action="CREATE" ID="8"><Connection Destination="CONCATENATE_REALS.VALUE2" Source="MOVING_AVERAGE.VALUE_MA"/></Request>
<Response ID="8" />
<Request Action="CREATE" ID="9"><Connection Destination="CONCATENATE_REALS.RUN" Source="MOVING_AVERAGE.RUN_O"/></Request>
<Response ID="9" />
<Request Action="CREATE" ID="10"><Connection Destination="MOVING_AVERAGE.RUN" Source="SENSOR_SIMULATOR.READ_O"/></Request>
<Response ID="10" />
<Request Action="CREATE" ID="11"><Connection Destination="CONCATENATE_REALS.VALUE1" Source="SENSOR_SIMULATOR.VALUE"/></Request>
<Response ID="11" />
<Request Action="START" ID="12"/>
<Response ID="12" />
```

Figure 4.6: Deployment output seen in the Console View.

4.4 Real-Time Collaboration

In *PTERO* the collaborative editing system was implemented using *Yjs* [23], a CRDT framework designed for distributed collaborative applications, allowing users to concurrently edit shared data structures while automatically resolving conflicts in a deterministic manner. *Yjs* was chosen in detriment of alternative CRDT implementations like *Automerge* due to its larger ecosystem. With built-in support for many text editor libraries [40], such as *CodeMirror*, adding real-time collaboration to the *Function Blocks Editor View* is very straightforward.

The graph structure representing the FBD application is synchronized through shared *Yjs* data structures. Operations such as creating function blocks, deleting connections, modifying inputs or moving nodes are propagated in real time to all connected clients. Each user therefore observes the same graph state with minimal delay. To integrate collaboration into the *Graph View*, the internal state changes of the graph were connected to the shared *Yjs* document. Whenever a local modification occurs in the graph, the corresponding operation is serialized and applied to the shared collaborative state. Likewise, remote updates received from other users are reflected back into the local graph editor instance.

4.5 Deployment

The deployment process is responsible for taking the application defined in the *Graph View* and bringing it to life across the configured *DINASORE* instances. It is broken down into a sequence of steps, each of which is described in the following subsections.

4.5.1 Synchronize with DINASOREs

The first step is to ensure that each *DINASORE* instance has access to the function blocks that it will need to run the application. This is done by sending all the *.fbt* and corresponding *.py* files in

the database, to every device. To achieve this the system can make use of the `synchronize.py` script that accompanies the *DINASORE* source code. A configuration file needs to be provided that tells the script where to send the function blocks to, like shown in Listing A.3. The server must have a valid *SSH* connection with each device in order to connect and send the files.

4.5.2 Send Commands

Once the function block files are in place, the server will send the necessary commands to each *DINASORE* instance to build and start the application. This includes creating the function block instances, writing the initial input values, establishing the connections between event and data ports and finally sending the start command to begin execution. The order in which these messages are issued is important, as connections can only be established between existing nodes. The following two subsections describe how the messages are constructed and what they contain.

4.5.2.1 Build messages

Before the messages can be sent, they need to be built into the format that *DINASORE* is expecting. It consists of a two part message: the first part refers to the configuration name, which by default will be *EMB_RES*, while the second is the actual message payload containing the wanted command. Each of these parts are preceded by a 3-byte header composed of a `0x50` byte followed by 2 bytes indicating the size of the corresponding content. A representation of this format is presented in Figure 4.7.

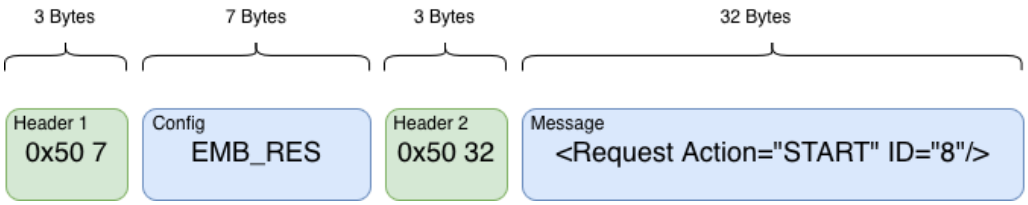


Figure 4.7: *DINASORE* communication format.

The *4diac_simulator.py* file, part of the *tests/* folder of *DINASORE*, implements this method in *Python*, for this prototype, it was necessary to implement it in *JavaScript*. The definitions of these methods, available for inspection, are presented in Section A.3.

4.5.2.2 Send Messages

Now that we have defined how to build the messages, we can explore the sequence of commands that need to be sent to each *DINASORE* instance. The server connects to each instance and transmits these messages sequentially. The order of message transmission is critical, as connections can only be established between existing nodes. The sequence is represented in Figure 4.8. A more technical and in depth analysis of the messages sent can be found in Section A.4.

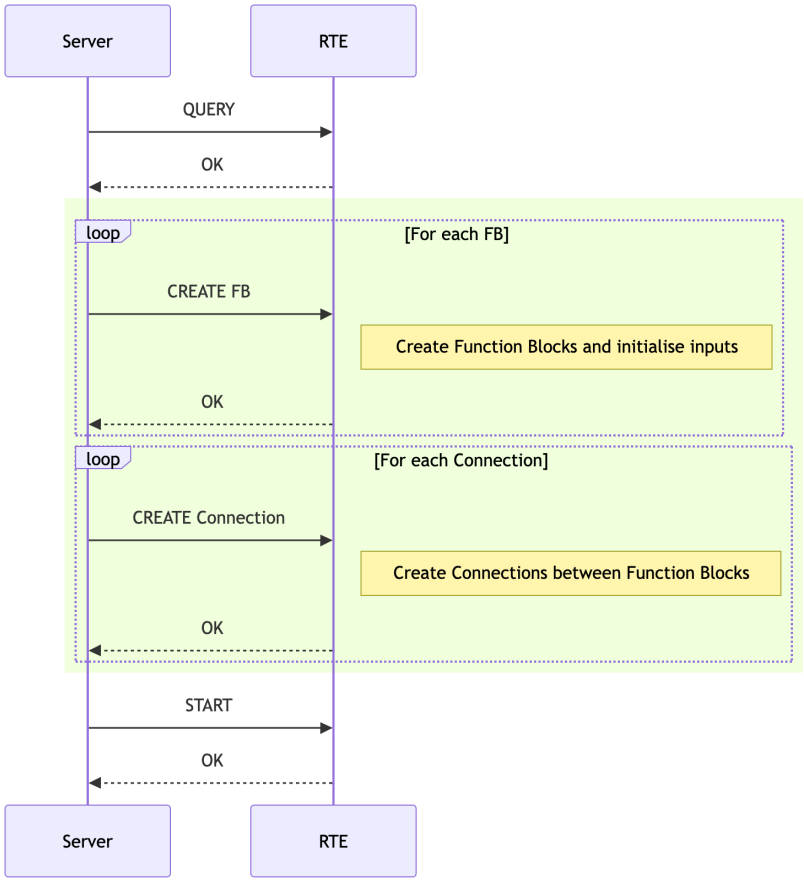


Figure 4.8: Overview of the communication sequence when deploying.

4.5.3 Console

To give the team visibility into the deployment process right from the IDE, without having to log into the server, a console is provided in the *Console View* that displays a live log of each action taken during deployment. There are two main things being logged:

- **Synchronize**: The output of running the *Python* script is displayed so that the team see that the *SSH* communication is being done properly.
- **Send Commands**: As commands are sent and responses are received, they are displayed, allowing the team to make sure each command is formulated correctly, being processed, and a response is issued.

4.5.4 Watch

Once the application is running, the values of the node outputs are continuously retrieved from the *DINASORE* instances and displayed in the *Graph View*, providing immediate feedback on the state of the running application. This is particularly useful for debugging and validating the behaviour of the application during development. In order to get these values, the IDE must tell

the *DINASORE* instances which output slots it wants to watch and then request the values. In this prototype, for simplicity, all data outputs are watched, and a watch creation request must be sent for each.

Figure 4.9 represents the sequence that the messages need to be sent. Details about these requests and the associated responses can be found in Section A.5. When the watch values are retrieved, they are displayed on the right side of the corresponding output slots, as demonstrated in Figure 4.10.

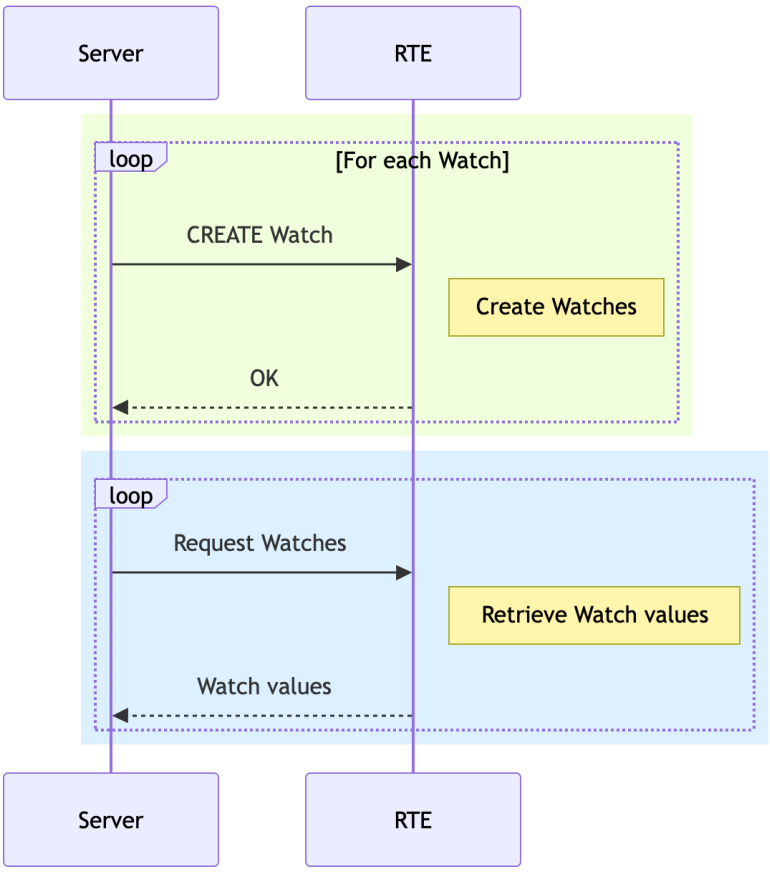


Figure 4.9: Overview of the communication sequence when watching.

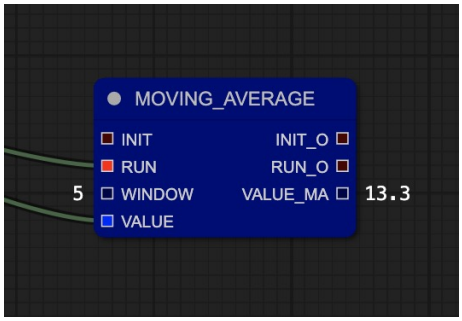


Figure 4.10: Watching an output slot in the *Graph View*.

Chapter 5

Validation

The purpose of this chapter is to evaluate whether the proposed solution achieves the proposed objectives, effectively addressing the gap previously identified. The evaluation combines objective measurements of system behaviour with subjective feedback collected from users. Together, these perspectives provide a comprehensive assessment of the prototype's effectiveness as an *IEC 61499* development environment. The hypothesis that guided this work is to be validated, and the research questions answered.

The evaluation combined both quantitative and qualitative methods, assessing objective performance characteristics alongside subjective user experience. Five complementary areas make up this validation. Functional validation verifies that the prototype correctly supports the modelling, deployment and monitoring of *IEC 61499* applications. Performance evaluation analyses workflow efficiency and compares key tasks against existing tools. Collaboration evaluation examines the benefits of the real-time collaborative architecture and contrasts it with traditional version control workflows. Usability evaluation focuses on user perception and experience, including a study of existing *IEC 61499* tooling. Finally, the results are discussed in the context of the research objectives and identified limitations.

5.1 Experimental Setup

A common test application was used throughout the validation process. The same application was implemented using both *PTERO* and *4DIAC-IDE*, allowing direct comparison between environments.

5.1.1 Hardware

The following hardware devices were available and used throughout the evaluation phase of this work:

1. **M1 MacBook Pro** (16 GB RAM, 2020, macOS 26): This machine served as the primary development and testing platform for *PTERO*. Being the development machine, it hosted

the local development server during iterative testing cycles and was used to evaluate the responsiveness of the collaborative editing interface, the correctness of graph synchronization and the performance of the *Yjs* based real-time collaboration layer under local conditions.

2. **ASUS TUF Gaming F16** (32 GB RAM, 2024, Windows 11): This machine was employed primarily for evaluating the performance of *4DIAC-IDE*.

3. **Raspberry Pi 3** (1 GB RAM, Raspberry Pi OS): This device acted as a permanently running, remotely accessible server throughout the evaluation period. It hosted a persistent instance of the *PTERO* server as well as an instance of *DINASORE*. Its limited memory and processing capacity made it the most representative of real-world target environments and its inclusion in the testing structure was instrumental in establishing the practical boundaries of the collaboration server.

5.1.2 Available Resources

The following resources were made available in the testing environments:

- One *DINASORE* RTE.
- Three function block definitions obtained from the official *DINASORE* tutorial *'Sensorization "Hello World!"'* [7]:
  - `SENSOR_SIMULATOR.fbt` and `SENSOR_SIMULATOR.py`

    Simulates a sensor read by emitting periodic events and producing simulated data.
  - `MOVING_AVERAGE.fbt` and `MOVING_AVERAGE.py`

    Calculates the moving average.
  - `CONCATENATE_REALS.fbt` and `CONCATENATE_REALS.py`

    Concatenates data into a string separated by a semicolon.

The selected function blocks provide a representative example of a small *IEC 61499* application while remaining sufficiently simple for controlled testing.

5.1.3 Test Application

The application shown in Figure 5.1 was implemented and deployed during the validation process. It consists of a sensor simulation function block whose output is processed by a moving average filter before being formatted for display.

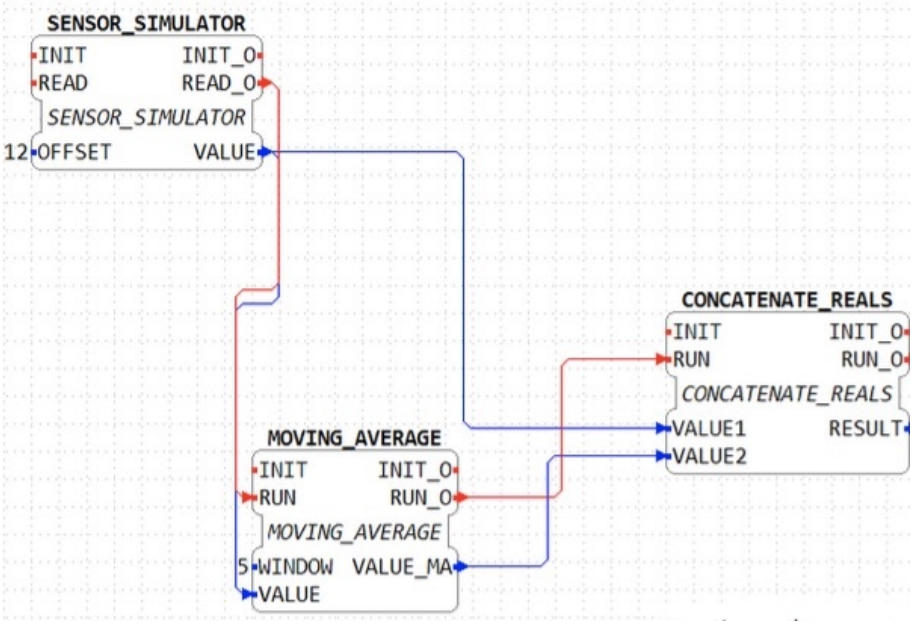


Figure 5.1: Test application implemented in *4DIAC-IDE*.

5.1.4 Expected Behaviour

After deployment, the application should execute correctly on the target *DINASORE* instance and continuously produce output values. These values should be observable through the monitoring capabilities provided by the IDE.

5.2 Functional Validation

PTERO was subjected to functional tests to confirm that all core features behave as specified. It was tested by implementing and deploying the test program defined in Section 5.1. As a control, the same was performed in *4DIAC-IDE*. The steps taken are described in Table 5.1.

Table 5.1: Workflow Comparison Between *4DIAC-IDE* and *PTERO*.

Step	4DIAC-IDE	PTERO
Configured available *DINASORE*	In the *System Configuration* tab, added a device and set its IP address and the port where *DINASORE* is running, then connected it to the *Ethernet* tunnel.	In the *Config* tab, added a device and set its IP address and the port where *DINASORE* is running, as well as the SSH details of the device which will be used to sync the function blocks files.
Imported the function block files	Copied and pasted the given files to `4diac-ide/typelibrary/`.	Copied and pasted the given files in the *Function Blocks* tab.

Table 5.1: Workflow Comparison Between *4DIAC-IDE* and *PTERO*.

Step	4DIAC-IDE	PTERO
Designed the flow	In the *App View*, dragged and dropped the wanted function blocks, then connected the event and data slots. Double-clicked input slots to assign input value.	In the *Graph* tab, right-clicked and selected the wanted function blocks to add them, then connected the event and data slots. Right-clicked nodes to assign input value.
Mapped to *DINASORE*	Right-clicked nodes to map them to the device. They became color coded to show which device they were mapped to.	Right-clicked nodes to map them to the device. They became color coded to show which device they were mapped to.
Synced files with *DINASORE*	Called the `synchronize.py` manually, feeding it the needed SSH details.	This was done automatically and without any needed setup when deploying.
Deployed	Pressed the *Deploy* button. The sent commands and associated responses were displayed in a console.	Pressed the *Deploy* button. The sent commands and associated responses were displayed in a console.
Monitored the output	The values were displayed on the right side of the output data slots.	The values were displayed on the right side of the output data slots.

Besides verifying that the output values are being monitored correctly, following the *Deployed* step, the commands displayed in the console were directly compared between IDEs. It was verified that both were sending exactly the same commands, the headers as well as the contents were exactly the same and the responses were no different. These results confirmed that the prototype correctly handles the development of *IEC 61499* programs.

While the two workflows are broadly equivalent in the number of steps required, they differ in several procedural details. Configuring a device in *PTERO* requires additional SSH credentials upfront, a step absent in *4DIAC-IDE*, however, this overhead is offset later in the workflow, as file synchronization with *DINASORE* is a manual command line operation in *4DIAC-IDE*, while it is handled automatically by *PTERO* at deploy time. Similarly, importing function block files is more straightforward in *PTERO*, which accepts files directly through the interface, whereas *4DIAC-IDE* requires manual placement into a specific directory in the filesystem. These differences reflect a deliberate design philosophy in *PTERO* of consolidating configuration complexity into the tool itself.

5.3 Performance Evaluation

This solution not only aims to execute the same core workloads as *4DIAC-IDE*, but also introduces architectural capabilities and design choices that justify its development. To assess its efficacy, the system's performance is evaluated against four primary metrics:

- **Installation Complexity**: The technical overhead and environmental constraints required to set up the development environment.
- **Startup Latency**: The temporal duration from initialization to full operational readiness.
- **Configuration Overhead**: The number of manual steps required to configure the system.
- **Deployment Flexibility**: The flexibility that the deployment system allows.

Taking these metrics into account, each step of development was tested and compared. The listed tasks in Table 5.2 are the ones that demonstrated to be significantly different between solutions, others were analysed but proved to not be worth the discussion.

Table 5.2: Performance Evaluation comparison.

Task	**4DIAC-IDE**	**PTERO**
Install	Installed the desktop app. There was no compatible version for macOS that supports *DINASORE* [7].	Installed a *Node.js* server and ran command line commands to set it up. Worked in all three major operating systems tested.
Open	Average of 4 seconds to boot the application in the *ASUS TUF Gaming F16*. The startup overhead is inherent to the *Eclipse* based desktop architecture underlying this tool.	Average of less than a second to load webpage. The use of lightweight libraries and the move of heavy workloads to the server allow the page to load in an instant.
Create and edit function block files	While it does have a way to create and edit the *.fbt* files inside the application, there is no built-in *Python* text editor.	Has a text editor for both types of files.

Table 5.2: Performance Evaluation comparison.

Task	**4DIAC-IDE**	**PTERO**
Collaboration	An external periodic version control system like *Git* needed to be used. Structured data in formats like *XML* is not handled properly by these types of version control system, which results in frequent merge conflicts [3]. The following test was conducted: 1. A project has a graph with 1 function block added. 2 users are working on the graph. 2. User #1 deletes the function block. 3. User #2 modifies the position of the function block. 4. User #1 commits and pushes. 5. User #2 tries to pull the changes but gets a merge conflict.	Has a real-time collaboration system. Any change made was immediately propagated to every client and resolved by the CRDT, ensuring no conflicts needed to be handled manually and damage is minimized.
Synchronizing files with the *DINASOREs*	Required to manually use an external script before deployment.	The server itself connected and synchronized the files with the *DINASOREs*. This happens automatically when attempting to deploy, ensuring that deployments never fail due to the human error of forgetting to sync the files.
Remote deployment	Every IDE needed to have a valid connection with each available *DINASORE*, as well as *SSH* access. This would be increasingly complicated if the user wanted to make a deployment while in a different network.	By having the server as the middleware, only it was exposed for the users to connect to. With only the server permanently having direct connection with the *DINASORE* devices, no individual handling was needed.

The **Install** and **Open** tasks evaluated installation complexity and startup latency. While *4DIAC-IDE* offers a straightforward local desktop installer, *PTERO* trades this for a one-time backend setup that guarantees universal cross-platform access via standard web browsers. Furthermore, by offloading computationally heavy operations to the backend, *PTERO* reduced client-side startup latency from 4 seconds down to under 1 second, fulfilling the objective for immediate operational readiness.

The remaining tasks evaluated configuration overhead and deployment flexibility. *4DIAC-IDE* introduced operational friction by requiring external text editors for script writing, manual script execution for synchronization and asynchronous version control operations. It also requires complex network setups since every engineering workstation must maintain individual direct connections to each target RTE. Conversely, *PTERO* minimizes configuration overhead by unifying file editing, automating file synchronization and introducing the concept a CRDT. By utilizing the server as a middleware, *PTERO* dramatically improved deployment flexibility, as clients only need to authenticate with the centralized server, not the RTEs.

5.4 Collaboration Evaluation

Nicolaescu *et al.* [23] establish that *Yjs* is able to handle an average of 100 actions per millisecond which is obviously good enough for this use case, as it does not require millisecond precision. Taking connection with the remote server into account, a test was devised to understand how well the *Raspberry Pi* could handle multiple users at the same time. A script was created that initialized a set number of clients, connected them to the remote server and sent a change that would get propagated across the clients. By measuring the time that would take for the clients to receive the change, the chart in Figure 5.2 was obtained. The results demonstrated that even a resource constrained machine accessed over a network can serve as an adequate collaboration server for a small team, however, the *Raspberry Pi* deployment was found to be unsuitable beyond approximately 240 simultaneous clients, at which point the hardware prevented additional connections from being established reliably.

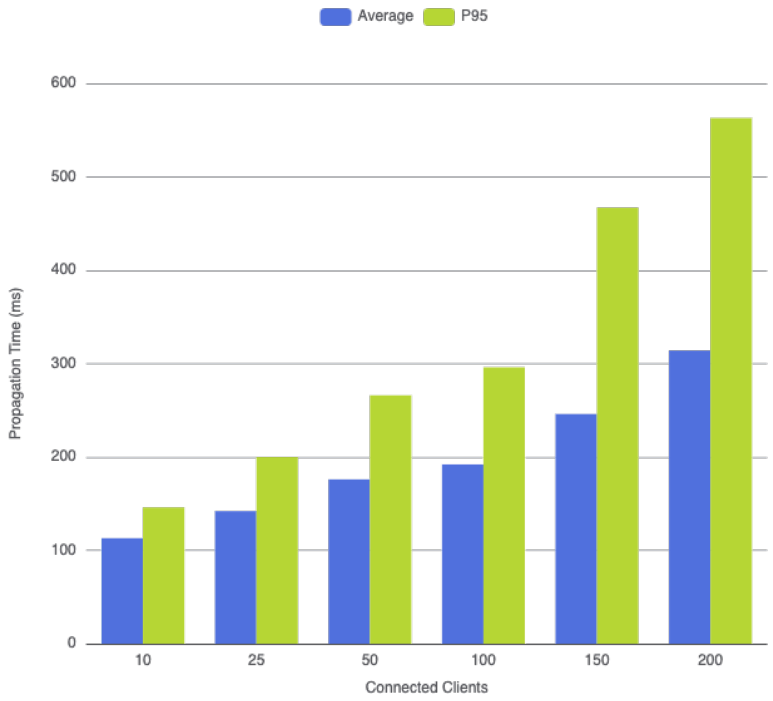


Figure 5.2: Performance results of *Yjs* running in the *Raspberry Pi*.

The same test was performed with a server running on the *M1 MacBook Pro* and with the clients also in the same machine, removing any delay resulting from network latency and in a much more powerful machine. It was possible to connect up to 3000 clients with an average propagation time of 65 ms and a P95 of 221 ms, proving that if there is a need for hundreds of clients to be connected at once, the solution is able to scale with the hardware to accommodate it.

5.5 Usability Evaluation

Two questionnaires were designed and given to participants. One focused on the user experience of working with *4DIAC-IDE* as the underlying development environment for *IEC 61499*. The other asked both for a usability opinion on *PTERO*, as well as a direct comparison between these two IDEs. The participants of the questionnaires were:

- 11 Students from Master in Electrical and Computer Engineering (MEEC) at Faculdade de Engenharia da Universidade to Porto (FEUP), enrolled in the course of Information Systems (SINF), in which, over the course of a semester, they were tasked with using *4DIAC-IDE* for modelling FBD programs with *DINASORE* as the RTE. Having completed a full semester of work on this environment, these students possessed sufficient practical familiarity with the tool to provide informed usability assessments and represent a realistic profile of future *PTERO* adopters in academic and educational contexts.

- 5 Students at FEUP, working on Master's dissertations at Research Center for Systems and Technologies (SYSTEC) on topics directly related to *4DIAC-IDE* and *DINASORE*. Their deeper technical engagement with the ecosystem made them particularly well positioned to evaluate the correctness and completeness of *PTERO*'s workflow.

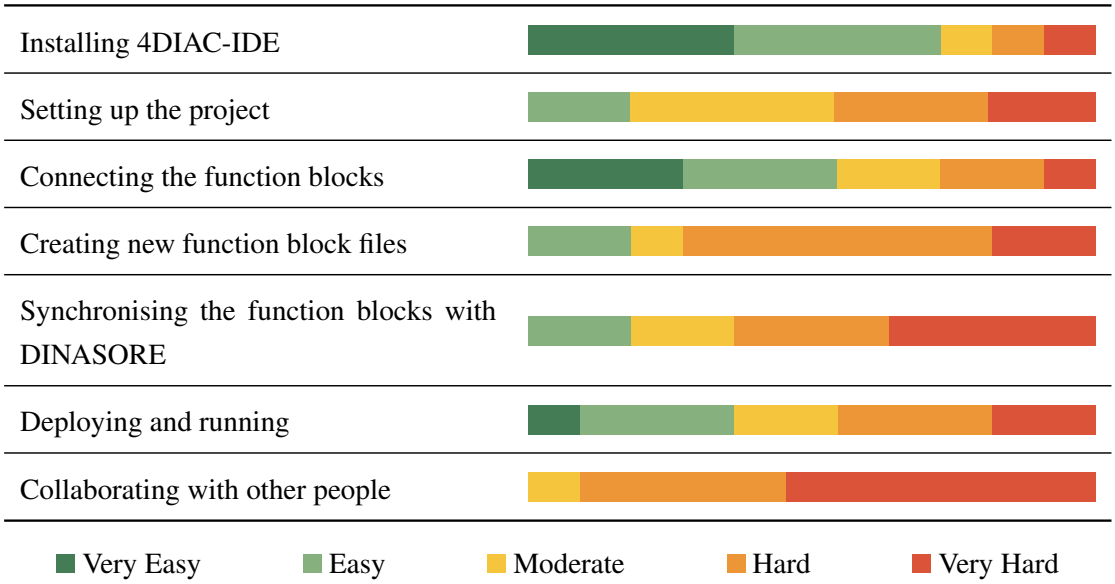
5.5.1 4DIAC-IDE Usability

A usability study on *4DIAC-IDE* has already been conducted by Weber, Zoitl, and HuBmann [39], but given as this new proposed solution introduces new concepts and expectations, a new usability study was devised to understand how this tool fairs in the specific fields that *PTERO* sets out to innovate in. Next follows the results of this new study.

Difficulty

With the use of a Likert [15] table, the difficulty of doing several different tasks using this IDE was analysed. Ranging from *Very Easy* to *Very Hard*, this section had the intent of understanding how easy and straightforward each step of *IEC 61499* development was. If a specific step was difficult in *4DIAC-IDE*, it will be interesting to see how it compares with *PTERO*. The results are displayed in Table 5.3.

Table 5.3: Perceived difficulty of tasks in *4DIAC-IDE*.



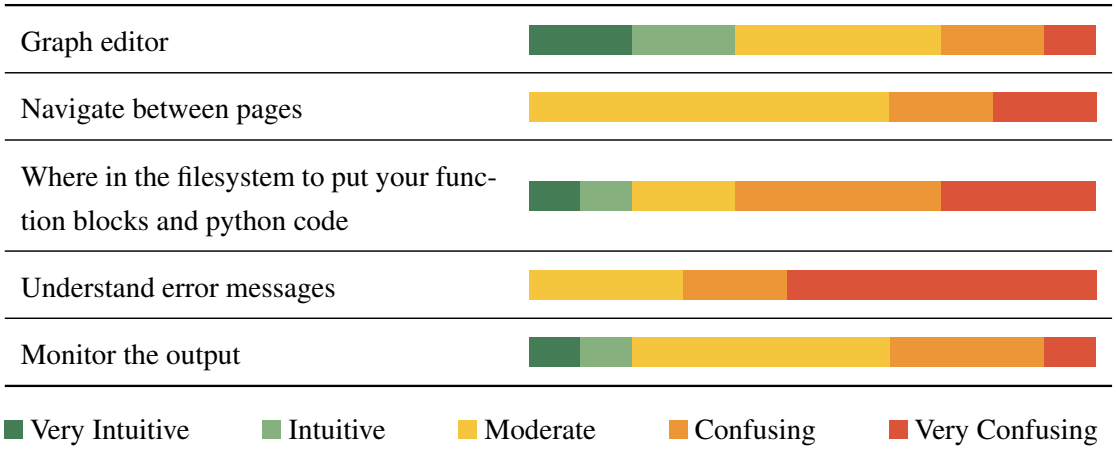
While initial installation and graphical operations showed good to moderate usability, other tasks were not so well reviewed. The most critical usability barriers manifested during distributed orchestration and collaboration. Manually synchronizing function blocks with *DINASORE* was rated as *Hard* or *Very Hard* by 63.7% of participants, reflecting the operational risks of relying

on external scripts. Furthermore, group collaboration yielded the most severe negative consensus, with 90.9% of respondents designating it as *Hard* or *Very Hard* (including a 54.5% absolute majority for *Very Hard*).

Intuitiveness

Next, another Likert table, this time on how intuitive it was to understand each part of the IDE. This way the design decisions of *4DIAC-IDE* can be critically analysed. The results are displayed in Table 5.4.

Table 5.4: Perceived intuitiveness of key parts of *4DIAC-IDE*.



The graph editor and the system for monitoring the output received decent evaluations, leaning primarily towards a *Moderate* rating. This suggests that the basic, traditional paradigms of visual component wiring are generally well-understood.

In contrast, structural navigation and file handling caused noticeable confusion. Not a single participant found that moving across views was intuitive. This ambiguity extends to file management, where 63.7% of respondents struggled to determine exactly where in the filesystem to place custom function block files, marking a clear disconnect between the application and the host directory structure.

Open Feedback

The questionnaire concluded with open-ended questions that provided additional insight into the user experience of working with *4DIAC-IDE*.

What did you like most about using 4DIAC-IDE? The most frequently mentioned positive aspect was the modularity of programming with function blocks, which made the overall structure of programs easier to understand.

> "The files were simple to implement in .fbt and python, so the logic could be easily done prior and then sequentially implement our ideas in code and then route them."

> "I highly appreciated the modularity of the system, specifically the ability to design independent data manipulation pipelines for each individual machine. This decentralized approach feels highly relevant and closely mimics real-world industrial scenarios."

What were the biggest challenges you faced? The most common difficulties were related to deployment, management of function block files and collaboration. Participants reported confusion regarding where files needed to be placed in the filesystem and how synchronization with *DINASORE* should be performed. As SINF students had been tasked over the semester to work in groups, the collaboration issues were a recurring theme in their answers. It is also noteworthy that the average response length increased substantially, from 108 characters in the previous question to 510 characters in this one. This suggests that participants had considerably more detailed feedback when discussing the challenges, often providing detailed explanations and examples.

> "Collaborating with colleagues was very hard, because of the management of file versions. Only one could be working at a time, and we could only change the file after it has uploaded on GitHub."

> "Connecting to DINASORE and understanding that there are changes to make in both file systems."

> "(...) Vague Error Messages: Debugging can be quite difficult due to a lack of specificity in error messages. (...) 4DIAC-IDE occasionally rejects the deployment of a project without explaining exactly why, sometimes only providing the name of an input variable without further context."

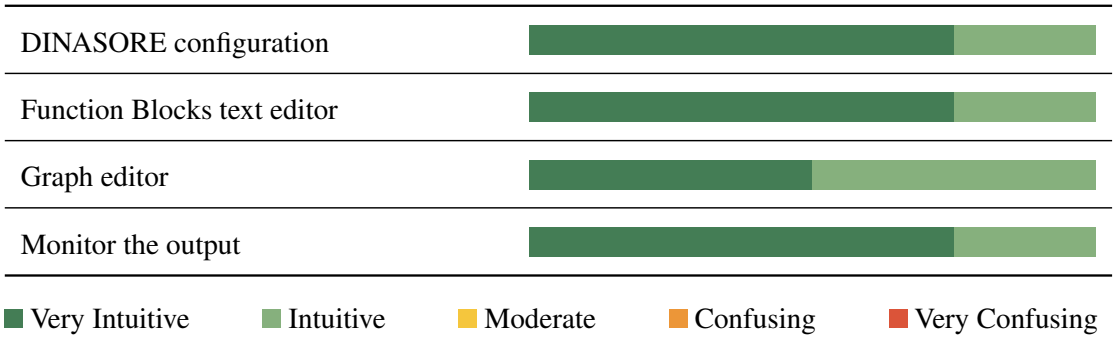
5.5.2 PTERO Usability

Following the user evaluation of *4DIAC-IDE*, the same set of participants was asked to recreate the previously defined *Test Application* using *PTERO*. To allow for more flexibility the *Raspberry Pi* hosted the server in a publicly accessible address, allowing participants to test the IDE remotely. Further details about the validation website can be found in Chapter B.

Intuitiveness

As before, a Likert table is used to evaluate how intuitive it was to understand each part of *PTERO*. This assessment aims to determine if the web design and unified workflow can result in a high perceived intuitiveness. The results are displayed in Table 5.5.

Table 5.5: Perceived intuitiveness of key parts of *PTERO*.



These results reveal a definitive positive consensus across all evaluated categories that participants find the workflow of *PTERO* to be intuitive. This represents a complete reversal of the confusion observed in the *4DIAC-IDE* study. The use of web technologies have allowed for a more user-friendly IDE.

System Usability Scale

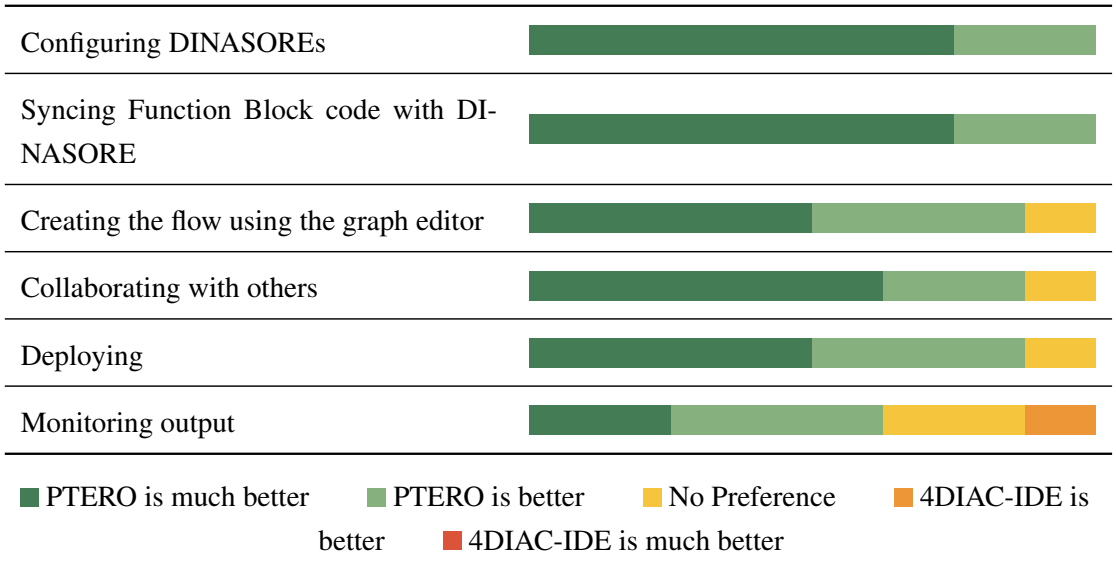
To obtain a globally standardized and quantifiable metric of *PTERO*'s overall user experience, the study incorporated the System Usability Scale (SUS) [17]. The SUS is a reliable scale where the raw point responses are aggregated and a score from 0 to 100 is calculated.

PTERO achieved an average SUS score of **88.75**. According to empirical usability benchmarks [16], this score positions *PTERO* well above the 95th percentile of all analysed applications. This outstanding evaluation confirms that *PTERO* is an extremely user-friendly solution.

5.5.3 Comparison

To provide a complete synthesis of the usability evaluation, the study concluded with a direct comparative survey. The core objective of this is to establish a clear relative performance delta, isolating whether *PTERO*'s remote web-based architecture delivers a measurably superior user experience when executing identical IEC 61499 engineering tasks. The comparative results are illustrated in Table 5.6.

Table 5.6: Direct comparison of usability of *PTERO* and *4DIAC-IDE*.



These results confirm the absolute preference of the participants for *PTERO* over *4DIAC-IDE* across all evaluated tasks, validating the success of the proposed system's architectural innovations.

The most substantial margins of improvement were achieved in target device configuration and orchestration. For both configuring *DINASOREs* and syncing the function block files, an absolute 100% consensus favoured *PTERO*. This flawless evaluation confirms that the middleware architecture that the server adopts is a great fit with this ecosystem.

Open Feedback

What do you like/dislike about a remote web-based solution like PTERO when compared to a desktop app like 4DIAC-IDE? Responses to this question were predominantly positive. Participants highlighted the reduced setup requirements, improved accessibility and simpler user interface provided by the web-based approach. Several respondents also noted that integrating the `.fbt` and `.py` editors into a single environment simplified the development workflow. While some usability improvements were suggested, participants considered PTERO easier to use than *4DIAC-IDE*.

> "Easier to set up for group work. Accessible from any PC."

> "PTERO seems easier to set up and overall use. Loved the feature of writing the .fbt and .py files on the same tab."

> "PTERO is an easy to use and very intuitive platform that surpasses in user-friendliness the 4DIAC-IDE when compared to its complexity and function block management."

5.6 Discussion

The evaluation results indicate that the proposed solution successfully achieves functional parity with the established *4DIAC-IDE* while introducing architectural and usability improvements that address several of the limitations identified in traditional *IEC 61499* development environments.

The functional validation confirmed that *PTERO* correctly supports the complete development workflow, including configuration, modelling, deployment and monitoring of *IEC 61499* applications. Both systems generated identical deployment commands and produced equivalent runtime behaviour in *DINASORE*. This equivalence is an important result, as it demonstrates that the proposed abstraction layer does not compromise correctness or compatibility with existing tooling. Instead, it preserves interoperability while offering a different interaction model.

The performance evaluation further complements these findings by highlighting how the architectural decisions in *PTERO* translate into measurable workflow improvements, confirming the **Hypothesis**. While both systems are capable of executing equivalent *IEC 61499* applications, *PTERO* reduces the number of manual steps required across key development tasks such as project setup, synchronization and deployment. A central factor behind this improvement is the tighter integration between the different components of the development pipeline. In *4DIAC-IDE*, the development workflow is distributed across multiple loosely connected tools and processes, requiring explicit user actions to maintain consistency between the modelling environment, local file system and remote runtime. In contrast, *PTERO* internalizes these responsibilities within the platform itself, allowing project state, collaboration and deployment logic to be handled automatically as part of a single unified system, answering the **RQ1**.

The usability questionnaires provide strong evidence that a web-based implementation can offer a significantly improved user experience compared to *4DIAC-IDE*. Participants consistently reported higher levels of perceived ease of use, intuitiveness and overall satisfaction when interacting with *PTERO*, providing rest to the **RQ2**. This improvement can be attributed not only to interface design choices, but also to the inherent advantages of web technologies in this context.

Furthermore, it was observed that the integration of all development tasks into a single unified interface contributes to a reduction in cognitive load. Unlike *4DIAC-IDE*, where users must switch between multiple tools and external processes (such as file system management and synchronization scripts), the web-based environment consolidates modelling, editing, deployment and monitoring into a coherent workflow. The results also suggest that responsiveness and perceived simplicity play an important role in user satisfaction.

Overall, the findings indicate that the adoption of web technologies is not merely a change in deployment platform, but a design decision that directly influences usability by simplifying access, unifying workflows and reducing structural complexity for end users.

Chapter 6

Conclusion

This dissertation set out to investigate whether modern web technologies and the shift to a middleware driven paradigm could meaningfully improve the development, deployment and operation of *IEC 61499*-compliant distributed automation systems. The answer, backed up by functional, performance, collaboration and usability evaluations, is affirmative.

6.1 Contributions and Key Findings

The central contribution of this work is *PTERO*, a web-based IDE for *IEC 61499* function block development. Rather than replicating the local-first architecture that characterizes existing tools, *PTERO* adopts a centralized server model in which the browser serves as the sole client-side requirement. This architectural decision has other benefits, such as eliminating per workstation installation and configuration, centralizing the deployment state and removing the need for individual workstations to maintain direct network access to target RTEs.

Beyond the architectural novelty, this dissertation contributes a concrete analysis of the gaps in the existing *IEC 61499* tooling ecosystem and demonstrates, through direct comparison with *4DIAC-IDE*, that those gaps can be addressed without sacrificing functional correctness. Both systems produced identical deployment command sequences and equivalent runtime behaviour in *DINASORE*, confirming that *PTERO*'s abstraction layer preserves full compatibility with the underlying standard and execution infrastructure.

The integration of real-time collaborative editing via a CRDT framework represents a further contribution of significance. The collaboration evaluation demonstrated that even resource-constrained hardware, such as the *Raspberry Pi* used throughout testing, can serve as a viable collaboration server for small engineering teams. On more capable hardware, the system scaled reliably to thousands of concurrent clients, validating the architectural scalability of the approach. This stands in contrast to the version control workflows that *4DIAC-IDE* teams must rely upon, which were consistently identified by study participants as a source of friction, merge conflicts and lost productivity.

The usability evaluation reinforced these findings with strong empirical support. The results confirm that the architectural improvements introduced by the web-based model translate directly into a measurably better user experience.

Taken together, the contributions of this dissertation demonstrate that the adoption of web technologies for *IEC 61499* tooling is a design decision that fundamentally restructures the development workflow in ways capable of benefiting accessibility, collaboration and usability.

6.2 Impact in the Field

The findings of this work carry implications that extend beyond *PTERO* itself and offer insights that the wider *IEC 61499* and industrial automation tooling community may draw upon.

The most transferable lesson concerns the treatment of collaboration as an architectural concern rather than a feature to be added after the fact. The difficulties participants reported with *4DIAC-IDE*'s collaboration model arose not from a lack of effort on the part of its developers, but from the inherent limitations of a local-first architecture applied to structured data. Future IDEs that wish to support collaborative workflows would benefit from adopting CRDT based shared state models, synchronizing their shared document over a background service rather than through periodic file exchange.

6.3 Limitations and Future Work

Several directions present themselves as natural extensions of this work and its limitations. Firstly, it is important to acknowledge the boundaries of the current prototype. *PTERO* was validated against *DINASORE*. While the management protocol shared between *DINASORE* and *4DIAC FORTE* means that extension to other compliant runtimes is architecturally straightforward, it has not been empirically verified in this work. Furthermore, the current implementation does not yet support the full breadth of *IEC 61499* constructs which would be required for more complex industrial applications.

From a collaboration standpoint, the current implementation propagates all graph changes to all connected clients without any notion of user identity or access control. Introducing user authentication, role-based editing permissions and change attribution would make *PTERO* more appropriate for team environments where auditability and access management are required.

On the infrastructure side, the current deployment model assumes that the *PTERO* server has persistent access to all registered *DINASORE* instances. Future work could explore the use of containerized deployment [24] and integration with CI pipelines, enabling *PTERO* to participate in automated testing and continuous workflows for *IEC 61499* applications.

Finally, the usability study, while yielding strongly positive results, involved a participant group drawn exclusively from an academic context, which is not enough to draw conclusions about the tool's suitability for production deployments. A wider usability study conducted in an industrial setting, in which engineering teams adopt *PTERO* over the course of a real development

project, would provide much richer empirical evidence than the controlled evaluation performed here.

References

[1] *4diac IDE Website*. URL: https://eclipse.dev/4diac/4diac_ide/ (visited on 06/15/2026).

[2] Iskra Antonova and Idilia Batchkova. "Development of Multi-Agent Control Systems using UML/SysML". In: Apr. 2011. ISBN: 978-953-307-174-9. DOI: 10.5772/14602.

[3] E. Axelsson and Sérgio Batista. "Version Control of structured data: A study on different approaches in XML". In: 2015. URL: https://hdl.handle.net/20.500.12380/219831 (visited on 05/31/2026).

[4] *BaklavaJS*. URL: https://baklava.tech/ (visited on 06/15/2026).

[5] Zakaria Bencherki *et al.* "IEC 61499: A Key Enabler for Distributed Automation in Industry 4.0". In: *2024 International Conference on Electrical, Computer and Energy Technologies (ICECET*. July 2024, pp. 1–6. DOI: 10.1109/ICECET61485.2024.10698592. URL: https://ieeexplore.ieee.org/document/10698592 (visited on 06/11/2026).

[6] Young-Jo Cho *et al.* "Function Block Diagram Approach for Manufacturing Process Control Programming". en. In: *IFAC Proceedings Volumes* 26.2 (July 1993), pp. 461–464. ISSN: 14746670. DOI: 10.1016/S1474-6670(17)48510-7. URL: https://linkinghub.elsevier.com/retrieve/pii/S1474667017485107 (visited on 01/28/2026).

[7] *DIGI2-FEUP/dinasore wiki*. original-date: 2020-04-21T12:20:20Z. May 2026. URL: https://github.com/DIGI2-FEUP/dinasore (visited on 06/14/2026).

[8] Juliane Fischer *et al.* "Measuring the Overall Complexity of Graphical and Textual IEC 61131-3 Control Software". In: *IEEE Robotics and Automation Letters* 6.3 (July 2021), pp. 5784–5791. ISSN: 2377-3766, 2377-3774. DOI: 10.1109/LRA.2021.3084886. URL: https://ieeexplore.ieee.org/document/9444196/ (visited on 06/07/2026).

[9] Francisco Folgado *et al.* "Review of Industry 4.0 from the Perspective of Automation and Supervision Systems: Definitions, Architectures and Recent Trends". en. In: *Electronics* 13.4 (Feb. 2024), p. 782. ISSN: 2079-9292. DOI: 10.3390/electronics13040782. URL: https://www.mdpi.com/2079-9292/13/4/782 (visited on 06/11/2026).

[10] Marcelo V. Garcia *et al.* "Engineering tool to develop CPPS based on IEC-61499 and OPC UA for oil&gas process". en. In: *2017 IEEE 13th International Workshop on Factory Communication Systems (WFCS)*. Trondheim, Norway: IEEE, May 2017, pp. 1–9. ISBN: 978-1-5090-5788-7. DOI: 10.1109/WFCS.2017.7991969. URL: http://ieeexplore.ieee.org/document/7991969/ (visited on 01/30/2026).

[11] *GoJS*. en. URL: https://gojs.net/latest/ (visited on 06/15/2026).

[12] *IEC 61131-3*. en. Page Version ID: 1339669093. Feb. 2026. URL: https://en.wikipedia.org/w/index.php?title=IEC_61131-3&oldid=1339669093 (visited on 05/11/2026).

[13] Steven Kelly. “Collaborative Modelling with Version Control”. In: ed. by Martina Seidl and Steffen Zschaler. Vol. 10748. Book Title: Software Technologies: Applications and Foundations Series Title: Lecture Notes in Computer Science. Cham: Springer International Publishing, 2018, pp. 20–29. ISBN: 978-3-319-74729-3 978-3-319-74730-9. DOI: 10.1007/978-3-319-74730-9_3. URL: http://link.springer.com/10.1007/978-3-319-74730-9_3 (visited on 06/04/2026).

[14] Martin Kleppmann and Alastair R. Beresford. “A Conflict-Free Replicated JSON Datatype”. In: *IEEE Transactions on Parallel and Distributed Systems* 28.10 (Oct. 2017), pp. 2733–2746. ISSN: 1045-9219. DOI: 10.1109/TPDS.2017.2697382. URL: http://ieeexplore.ieee.org/document/7909007/ (visited on 06/09/2026).

[15] Malcolm Koo and Shih-Wei Yang. “Likert-Type Scale”. en. In: *Encyclopedia* 5.1 (Feb. 2025), p. 18. ISSN: 2673-8392. DOI: 10.3390/encyclopedia5010018. URL: https://www.mdpi.com/2673-8392/5/1/18 (visited on 06/15/2026).

[16] James Lewis and Jeff Sauro. “Item Benchmarks for the System Usability Scale”. In: 13 (May 2018), pp. 158–167.

[17] James R. Lewis. “The System Usability Scale: Past, Present, and Future”. In: *International Journal of Human–Computer Interaction* 34.7 (July 2018), pp. 577–590. ISSN: 1044-7318. DOI: 10.1080/10447318.2018.1455307. URL: https://doi.org/10.1080/10447318.2018.1455307 (visited on 06/14/2026).

[18] Geng Liang. “A kind of implementation model based on generalized FBD for distributed intelligent control network”. In: *2008 Chinese Control and Decision Conference*. ISSN: 1948-9447. July 2008, pp. 411–414. DOI: 10.1109/CCDC.2008.4597341. URL: https://ieeexplore.ieee.org/document/4597341 (visited on 06/13/2026).

[19] *litegraph.js*. URL: https://github.com/jagenjo/litegraph.js/ (visited on 06/15/2026).

[20] *Low-code programming for event-driven applications : Node-RED*. URL: https://nodered.org/ (visited on 06/15/2026).

[21] Tuojian Lyu *et al.* “Towards Web Platform for Cloud-Based Virtual Commissioning of IEC 61499 Distributed Automation Systems”. en. In: *2024 IEEE 29th International Conference on Emerging Technologies and Factory Automation (ETFA)*. Padova, Italy: IEEE, Sept. 2024, pp. 1–4. ISBN: 979-8-3503-6123-0. DOI: 10.1109/ETFA61755.2024.10710351. URL: https://ieeexplore.ieee.org/document/10710351/ (visited on 12/17/2025).

[22] László Monostori. “Cyber-physical Production Systems: Roots, Expectations and R&D Challenges”. en. In: *Procedia CIRP* 17 (2014), pp. 9–13. ISSN: 22128271. DOI: 10.1016/j.procir.2014.03.115. URL: https://linkinghub.elsevier.com/retrieve/pii/S2212827114003497 (visited on 01/28/2026).

[23] Petru Nicolaescu *et al.* “Yjs: A Framework for Near Real-Time P2P Shared Editing on Arbitrary Data Types”. en. In: ed. by Philipp Cimiano *et al.* Vol. 9114. Book Title: Engineering the Web in the Big Data Era Series Title: Lecture Notes in Computer Science. Cham: Springer International Publishing, 2015, pp. 675–678. ISBN: 978-3-319-19889-7 978-3-319-19890-3. DOI: 10.1007/978-3-319-19890-3_55. URL: http://link.springer.com/10.1007/978-3-319-19890-3_55 (visited on 06/09/2026).

[24] Nikolaos Nikolakis *et al.* “On a containerized approach for the dynamic planning and control of a cyber - physical production system”. In: *Robotics and Computer-Integrated Manufacturing* 64 (Aug. 2020), p. 101919. ISSN: 0736-5845. DOI: 10.1016/j.rcim.2019.101919. URL: https://www.sciencedirect.com/science/article/pii/S0736584519300183 (visited on 06/14/2026).

[25] *Node-Based UIs for React and Svelte*. en. URL: https://xyflow.com (visited on 06/15/2026).

[26] Linna Pang *et al.* “Formal verification of function blocks applied to IEC 61131-3”. en. In: *Science of Computer Programming* 113 (Dec. 2015), pp. 149–190. ISSN: 01676423. DOI: 10.1016/j.scico.2015.10.005. URL: https://linkinghub.elsevier.com/retrieve/pii/S0167642315002981 (visited on 01/29/2026).

[27] E. Pereira, João C. P. Reis, and G. Gonçalves. “DINASORE: A Dynamic Intelligent Reconfiguration Tool for Cyber-Physical Production Systems”. In: 2020. URL: http://ceur-ws.org/Vol-2739/#paper_9 (visited on 06/02/2026).

[28] Gonçalo Pinto *et al.* “Digital Factory: An Industrial Case Study for Function Block-Oriented Development”. en. In: *2023 IEEE 9th World Forum on Internet of Things (WF-IoT)*. Aveiro, Portugal: IEEE, Oct. 2023, pp. 01–06. ISBN: 979-8-3503-1161-7. DOI: 10.1109/WF-IoT58464.2023.10539486. URL: https://ieeexplore.ieee.org/document/10539486/ (visited on 01/29/2026).

[29] *Rete.js*. URL: https://rete.js.org/ (visited on 06/15/2026).

[30] Oana Rohat and Dan Popescu. “Remote web-based execution of IEC 61499 function blocks”. en. In: *Proceedings of the 2014 6th International Conference on Electronics, Computers and Artificial Intelligence (ECAI)*. Bucharest, Romania: IEEE, Oct. 2014, pp. 33–38. ISBN: 978-1-4799-5478-0 978-1-4799-5479-7. DOI: 10.1109/ECAI.2014.7090220. URL: http://ieeexplore.ieee.org/document/7090220/ (visited on 01/28/2026).

[31] Oliver Schmid, Agnes Lisowska Masson, and Béat Hirsbrunner. “Real-time collaboration through web applications: an introduction to the Toolkit for Web-based Interactive Collaborative Environments (TWICE)”. en. In: *Personal and Ubiquitous Computing* 18.5 (June 2014), pp. 1201–1211. ISSN: 1617-4909, 1617-4917. DOI: 10.1007/s00779-013-0729-0. URL: http://link.springer.com/10.1007/s00779-013-0729-0 (visited on 06/04/2026).

[32] Jero Soler. *Drawflow*. original-date: 2020-04-21T17:59:14Z. June 2026. URL: https://github.com/jerosoler/Drawflow (visited on 06/15/2026).

[33] Radimir Sorokin, Sandeep Patil, and Valeriy Vyatkin. “Novel development tool for IEC 61499 based on domain-specific languages”. en. In: *IFAC-PapersOnLine* 55.2 (2022), pp. 439–444. ISSN: 24058963. DOI: 10.1016/j.ifacol.2022.04.233. URL: https://linkinghub.elsevier.com/retrieve/pii/S2405896322002348 (visited on 01/28/2026).

[34] Radimir Sorokin and Valeriy Vyatkin. “Towards web-applications for domain-specific languages and development tools”. In: *2023 IEEE 2nd Industrial Electronics Society Annual On-Line Conference (ONCON)*. Dec. 2023, pp. 1–6. DOI: 10.1109/ONCON60463.2023.10431381. URL: https://ieeexplore.ieee.org/document/10431381 (visited on 06/13/2026).

[35] Haiyang Sun *et al.* “Reasoning about the Node.js Event Loop using Async Graphs”. In: *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. Feb. 2019, pp. 61–72. DOI: 10.1109/CGO.2019.8661173. URL: https://ieeexplore.ieee.org/document/8661173 (visited on 06/13/2026).

[36] Tomás Torres, Gil Gonçalves, and Rui Pinto. “Literature Review on Cloud-Based Service-Oriented Architecture for IEC 61499 Distributed Control Systems:” en. In: *Proceedings of the 15th International Conference on Cloud Computing and Services Science*. Porto, Portugal: SCITEPRESS - Science and Technology Publications, 2025, pp. 174–181. ISBN: 978-989-758-747-4. DOI: 10.5220/0013293400003950. URL: https://www.scitepress.org/DigitalLibrary/Link.aspx?doi=10.5220/0013293400003950 (visited on 01/28/2026).

[37] *VSCode Web*. URL: https://code.visualstudio.com/docs/setup/vscode-web (visited on 01/29/2026).

[38] Valeriy Vyatkin and Julien Chouinard. *On comparisons of the ISaGRAF implementation of IEC 61499 with FBDK and other implementations*. Journal Abbreviation: 6th IEEE International Conference on Industrial Informatics Pages: 294 Publication Title: 6th IEEE International Conference on Industrial Informatics. Aug. 2008. ISBN: 978-1-4244-2170-1. DOI: 10.1109/INDIN.2008.4618111.

[39] Thomas Weber, Alois Zoitl, and Heinrich HuBmann. “Usability of Development Tools: A CASE-Study”. en. In: *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. Munich, Germany: IEEE, Sept. 2019, pp. 228–235. ISBN: 978-1-7281-5125-0. DOI: 10.1109/MODELS-C.2019.00037. URL: https://ieeexplore.ieee.org/document/8904489/ (visited on 01/30/2026).

[40] *Yjs Docs*. en. Dec. 2023. URL: https://docs.yjs.dev/ecosystem/editor-bindings (visited on 06/09/2026).

[41] Alois Zoitl, Thomas Strasser, and Gerhard Ebenhofer. “Developing modular reusable IEC 61499 control applications with 4DIAC”. en. In: *2013 11th IEEE International Conference on Industrial Informatics (INDIN)*. Bochum, Germany: IEEE, July 2013, pp. 358–363. ISBN: 978-1-4799-0752-6. DOI: 10.1109/INDIN.2013.6622910. URL: http://ieeexplore.ieee.org/document/6622910/ (visited on 01/29/2026).

Appendix A

Further Details of the Implementation

This appendix contains implementation details that were not important enough to include in Chapter 4. Most of the content here are big listings that would break the flow of the chapter, so this place was devised for displaying them.

A.1 Function Block Files

Examples of the formats of the *.fbt* and *.py* files that make up a function block are represented in Listing A.1 and Listing A.2, respectively.

```
<FBType Name="NEW_FB_1">
  <InterfaceList>
    <EventInputs>
      <Event Name="INIT" Type="Event"/>
      <Event Name="READ" Type="Event">
        <With Var="VARIABLE_1"/>
        <With var="VARIABLE_2"/>
      </Event>
    </EventInputs>
    <EventOutputs>
      <Event Name="INIT_O" Type="Event"/>
      <Event Name="READ_O" Type="Event">
        <With Var="DATA_1"/>
      </Event>
    </EventOutputs>
    <InputVars>
      <VarDeclaration Name="VARIABLE_1" Type="STRING"/>
      <VarDeclaration Name="VARIABLE_2" Type="INT"/>
    </InputVars>
    <OutputVars>
      <VarDeclaration Name="DATA_1" Type="STRING"/>
    </OutputVars>
  </InterfaceList>
</FBType>
```

Listing A.1: Example *xml* definition of a function block

```
class NEW_FB_1:
  def __init__(self):
    pass

  def schedule(self, event_name, event_value, variable_1, variable_2, variable_3):
    if event_name == "INIT":
      return [event_value, None, ""]

    elif event_name == "READ":
      return [None, event_value, ""]
```

Listing A.2: Example *Python* code of a function block

A.2 Synchronization

Listing A.3 presents an example configuration that the `synchronize.py` script requires in order to send the files to the *DINASORE* devices.

```
{
  master-fbs-path: 'sync/fbs',
  dinasores: [
    {
      address: 'x.x.x.x',
      port: 22,
      username: 'root',
      password: 'dinasore',
      dinasore-path: '/root/dinasore'
    },
    {
      address: 'y.y.y.y',
      port: 22,
      username: 'root',
      password: 'dinasore',
      dinasore-path: '/root/dinasore'
    }
  ]
}
```

Listing A.3: Sync configuration

A.3 Communication

The previously mentioned methods to build the messages are replicated in Listings A.4 and A.5.

```python
def build_message(message_payload, configuration_name):
  # build the first part of the header
  hex_input = '{:04x}'.format(len(configuration_name))
  second_byte = int(hex_input[0:2], 16)
  third_byte = int(hex_input[2:4], 16)
  response_len = struct.pack('BB', second_byte, third_byte)
  response_header_1 = b''.join([b'\x50', response_len])

  # build the second part of the header
  hex_input = '{:04x}'.format(len(message_payload))
  second_byte = int(hex_input[0:2], 16)
  third_byte = int(hex_input[2:4], 16)
  response_len = struct.pack('BB', second_byte, third_byte)
  response_header_2 = b''.join([b'\x50', response_len])

  # join the 3 parts
  response = b''.join([response_header_1, configuration_name, response_header_2,
      message_payload])
  return response
```

Listing A.4: build_message() in Python

```javascript
function build_message(message_payload, configuration_name) {
  const configBuffer = Buffer.from(configuration_name, 'utf-8');
  const messageBuffer = Buffer.from(message_payload, 'utf-8');

  // Header 1: [0x50][len_hi][len_lo]
  const header1 = Buffer.alloc(3);
  header1[0] = 0x50;
  header1.writeUInt16BE(configBuffer.length, 1);

  // Header 2: [0x50][len_hi][len_lo]
  const header2 = Buffer.alloc(3);
  header2[0] = 0x50;
  header2.writeUInt16BE(messageBuffer.length, 1);

  return Buffer.concat([
    header1,
    configBuffer,
    header2,
    messageBuffer
  ]);
}
```

Listing A.5: build_message() in JavaScript

A.4 Send Messages

This section present a more detailed explanation of the commands sent during deployment. First, the two commands in Listing A.6 need to be sent to every device. A `QUERY` request and a `CREATE` request that will create an *EMB_RES* resource which will be automatically connected to the *INIT* port of all other function block nodes to initialize them. Both these commands will have an empty configuration name, as *EMB_RES* has not been created yet and therefore cannot be bounded to.

```
<Request Action="QUERY" ID="0">
  <FB Name="*" Type="*"/>
</Request>

<Request Action="CREATE" ID="1">
  <FB Name="EMB_RES" Type="EMB_RES"/>
</Request>
```

Listing A.6: Setup deployment commands

Next, depending on which nodes are mapped to each machine, it needs to `CREATE` all the mapped nodes and `WRITE` the initial input values with the commands in Listing A.7. The configuration name in these and all following commands will be "*EMB_RES*" as they need to be bound to it in order to be initialized and ran.

```
<Request Action="CREATE" ID="2">
  <FB Name="SENSOR_SIMULATOR" Type="SENSOR_SIMULATOR"/>
</Request>

<Request Action="WRITE" ID="3">
  <Connection Destination="SENSOR_SIMULATOR.OFFSET" Source="12"/>
</Request>

<Request Action="CREATE" ID="4">
  <FB Name="MOVING_AVERAGE" Type="MOVING_AVERAGE"/>
</Request>

<Request Action="WRITE" ID="5">
  <Connection Destination="MOVING_AVERAGE.WINDOW" Source="5"/>
</Request>
```

Listing A.7: `CREATE` nodes deployment commands

Then the connections are created between the mapped nodes. A `Source` a `Destination` must be provided, with the format `"NODE_NAME.SLOT_NAME"`, using the commands in Listing A.8.

```
<Request Action="CREATE" ID="6">
  <Connection Destination="SENSOR_SIMULATOR.READ_0" Source="MOVING_AVERAGE.RUN"/>
</Request>

<Request Action="CREATE" ID="7">
  <Connection Destination="SENSOR_SIMULATOR.VALUE" Source="MOVING_AVERAGE.VALUE"/>
</Request>
```

Listing A.8: `CREATE` links deployment commands

Finally, the `START` command in Listing A.9 can be issued, which will request for the start of the program execution. For this command the configuration name used will also be "*EMB_RES*" as it is this resource that will begin the execution of the program.

```
<Request Action="START" ID="8"/>
```

Listing A.9: `START` deployment command

After each request is sent, a response will be returned by the *DINASORE*. It is important for the IDE to listen for these responses and only send the next request once the response corresponding to the previous request is received. The responses have the format shown in Listing A.10. The `ID` of the response will be the same of the associated request. The format of the responses is the same as the requests and the configuration name that is used is empty.

```
<Response ID="8"/>
```

Listing A.10: `Response` after a deployment command is sent

A.5 Watch

Note in Listing A.11 that the `ID` is reset after deploying.

```
<Request ID="0" Action="CREATE">
  <Watch Source="SENSOR_SIMULATOR.VALUE" Destination="*" />
</Request>

<Request ID="1" Action="CREATE">
  <Watch Source="MOVING_AVERAGE.VALUE_MA" Destination="*" />
</Request>
```

Listing A.11: `CREATE` watches after deploying

After creating the watches, request can be sent in order to receive the watch values using the command in Listing A.12. This will return the response in Listing A.13, which by parsing the values, allows the prototype to display them in the *Graph View*.

```
<Request ID="2" Action="READ">
  <Watches/>
</Request>
```

Listing A.12: Request to `READ` watches

```
<Response ID="2">
  <Watches>
    <Resource name="EMB_RES">
      <FB name="SENSOR_SIMULATOR">
        <Port name="VALUE">
          <Data value="13.75" />
        </Port>
      </FB>
      <FB name="MOVING_AVERAGE">
        <Port name="VALUE">
          <Data value="13.3" />
        </Port>
      </FB>
    </Resource>
  </Watches>
</Response>
```

Listing A.13: Response to `READ` watches

Appendix B

Usability Validation Website

The website used for usability validation was hosted at https://ptero.pedro-beirao.eu, and consisted of an instructions page with the steps required to recreate the *Test Application* from Section 5.1.3 and the IDE frontend itself. The Figures B.1 to B.4 preserve the state of the website during evaluation.

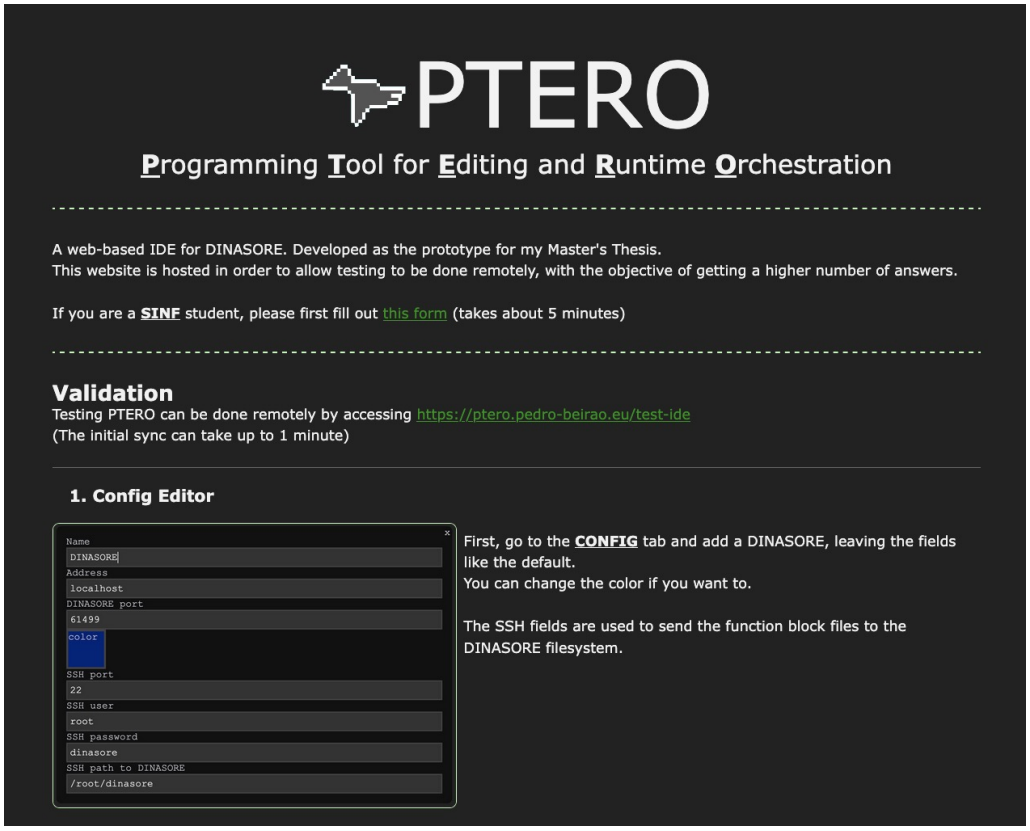


Figure B.1: Configuration instructions.

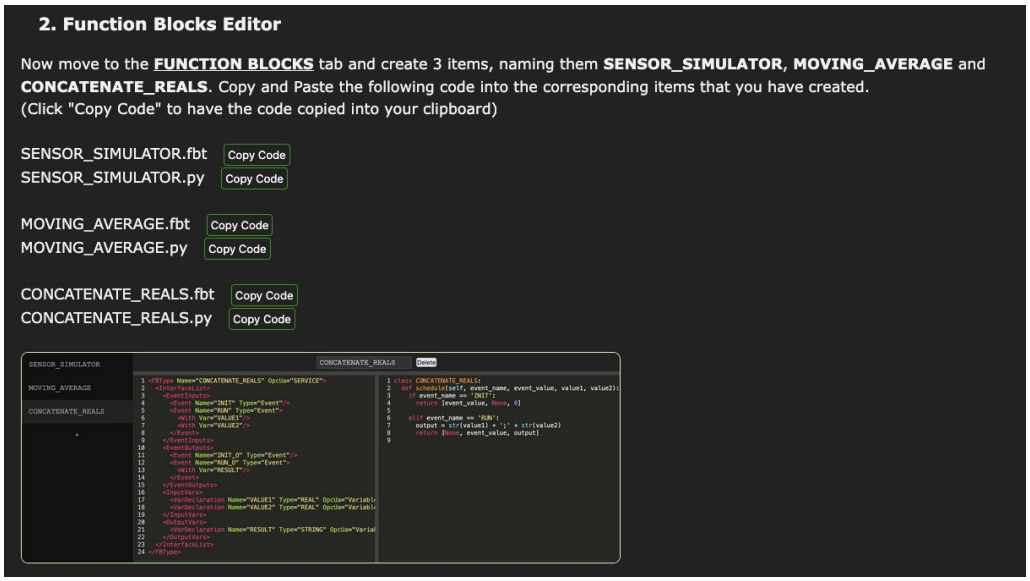


Figure B.2: Code editing instructions.

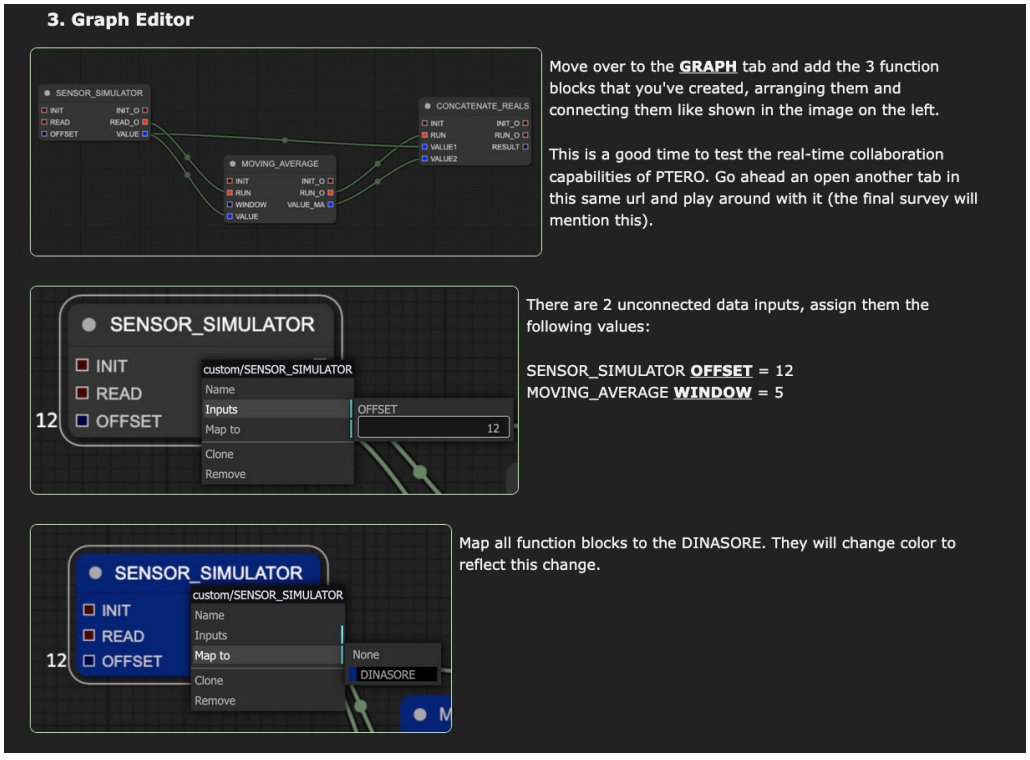


Figure B.3: Graph instructions.

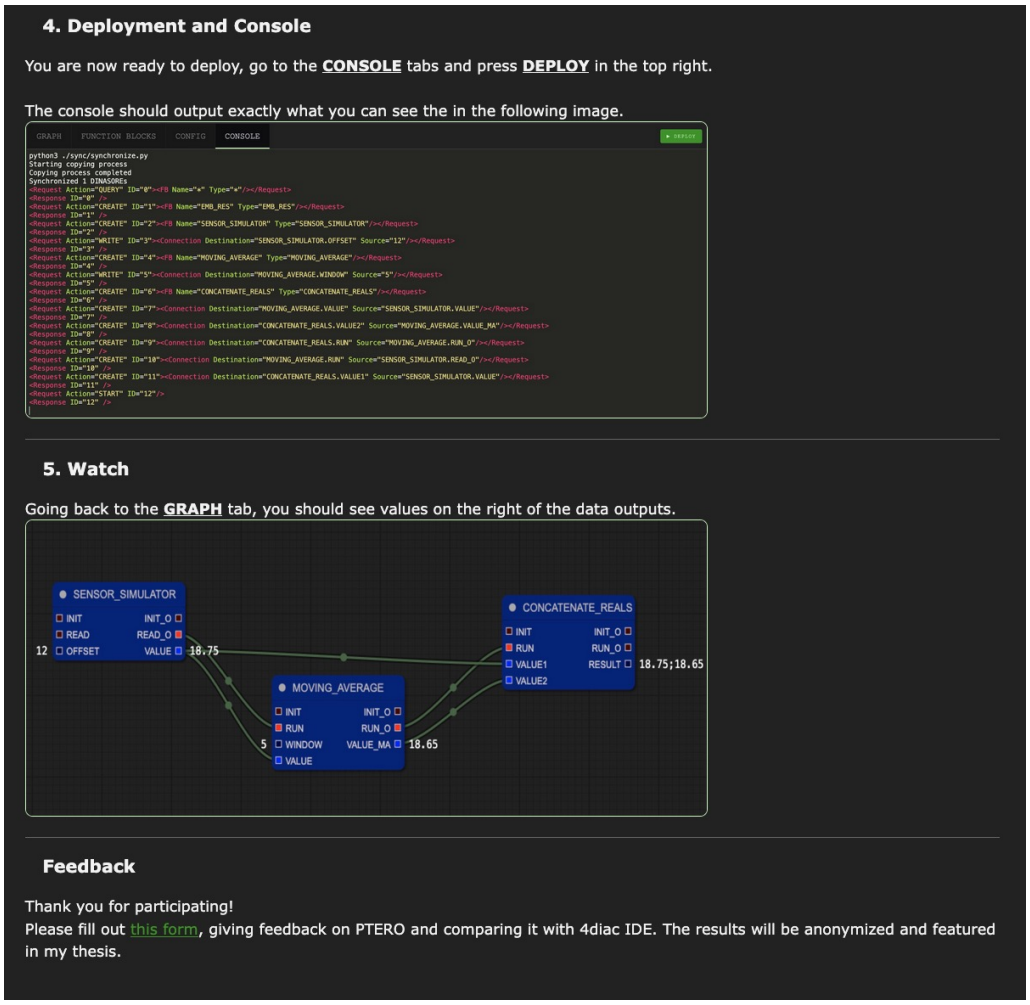


Figure B.4: Deployment and monitoring instructions.