FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Modeling Complex CPS with IEC61499 Composite FBs and DINASORE RTE

Gonçalo Luís Mendes Magalhães



Mestrado em Engenharia Eletrotécnica e de Computadores

Supervisor: Prof. Rui Pedro Ferreira Pinto

October 16, 2026

© Gonçalo Luís Mendes Magalhães, 2026

Modeling Complex CPS with IEC61499 Composite FBs and DINASORE RTE

Gonçalo Luís Mendes Magalhães

Mestrado em Engenharia Eletrotécnica e de Computadores

Approved in oral examination by the committee:

President: Prof. Mário Jorge Rodrigues de Sousa

Referee: Prof. José Fernando Lopes Barbosa

Referee: Prof. Rui Pedro Ferreira Pinto

October 16, 2026

Resumo

A quarta revolução industrial introduziu uma transformação profunda nos sistemas de automação industrial, marcada pela integração de sistemas ciberfísicos, inteligência distribuída e comunicação em tempo real entre plataformas heterogéneas. Embora estas tecnologias permitam maiores níveis de flexibilidade, adaptabilidade e eficiência, conduzem igualmente a um aumento significativo da complexidade dos sistemas, colocando desafios relevantes às arquiteturas de controlo centralizadas e aos modelos de software tradicionais. Neste contexto, a norma IEC 61499 emergiu como um referencial para o desenvolvimento de sistemas de controlo distribuído, promovendo a modularidade, a execução orientada a eventos e a independência de plataformas através do uso de Function Blocks. Entre estes, os Composite Function Blocks assumem particular relevância, ao possibilitar a composição hierárquica de funcionalidades e a modelação estruturada de comportamentos complexos, adequando-se de forma natural às necessidades dos sistemas ciberfísicos de produção. A presente dissertação analisa o papel dos Composite Function Blocks na modelação de arquiteturas complexas de sistemas ciberfísicos e avalia a sua implementação prática recorrendo ao Run-Time Environment DINASORE. Este ambiente de execução, desenvolvido em Python, estende o paradigma da IEC 61499 ao permitir a integração direta com bibliotecas modernas de processamento de dados, bem como a interoperabilidade nativa através de OPC UA. O trabalho visa avaliar a eficácia da abordagem baseada em Composite Function Blocks na gestão da complexidade dos sistemas e propor diretrizes de modelação que contribuam para o desenvolvimento de sistemas de controlo industriais mais modulares, escaláveis e adaptáveis aos requisitos da Indústria 4.0.

Os resultados demonstraram que a encapsulação em CFB não introduz sobrecarga computational mensurável em tempo de execução, reduzindo consistemente o tempo de deployment. Estabelece ainda que a reutilização especulativa baseada em memoização pode reduzir significativamente a carga computacional em workloads determinísticos e de baixa frequência, embora a sua eficácia diminua com o aumento da complexidade da pipeline. A partir destas conclusões, são derivadas diretrizes práticas de design para a utilização eficaz de CFBs em deployments edge com recursos limitados.

Palavras-chave: IEC 61499, Composite Function Blocks, Edge Computing, Sistemas Ciber-Físicos, DINASORE, Automação Distribuída, Memoização, Computação Especulativa, OPC UA, Indústria 4.0

Abstract

The fourth industrial revolution has introduced a paradigm shift in industrial automation, characterized by the integration of cyber-physical systems (CPS), distributed intelligence, and real-time communication across heterogeneous platforms. While these technologies enable higher flexibility, adaptability and efficiency, they also introduce a significant increase in system complexity, challenging traditional centralized control architectures and static software models. In this context, the IEC 61499 standard has emerged as a reference framework for the development of distributed industrial control systems, promoting modularity, event-driven execution, and platform-independent deployment through Function Blocks. Among these, Composite Function Blocks (CFBs) provide a hierarchical abstraction mechanism that enables the structured modeling of complex behaviors by composing simpler functional units, making them particularly suitable for cyber-physical production systems. This thesis investigates the role of CFBs in modeling complex CPS architectures and evaluates their practical implementation using the DINASORE Run-Time Environment (RTE). DINASORE extends the IEC 61499 execution paradigm by leveraging Python-based runtime capabilities, enabling seamless integration with modern data processing, as well as native OPC UA interoperability. The work aims to assess the effectiveness of CFB-based modeling in managing system complexity and to propose design guidelines for scalable, flexible, and intelligent industrial control systems aligned with Industry 4.0 requirements. The results demonstrate that CFB encapsulation introduces no measurable runtime computational overhead while consistently reducing deployment time. It also established that memoization-based speculative reuse can significantly reduce computation load under deterministic, low-frequency workloads, though its effectiveness degrades as pipeline complexity increases. From these findings, practical design guidelines are derived for the effective use of CFBs in resource-constrained edge deployments.

Keywords: IEC 61499, Composite Function Blocks, Edge Computing, Cyber-Physical Systems, DINASORE, Distributed Automation, Memoization, Speculative Computing, OPC UA, Industry 4.0

UN Sustainable Development Goals

The United Nations Sustainable Development Goals (SDGs) provide a global framework to achieve 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 thesis contributes to two of these goals, by exploring modular, reconfigurable approaches to distributed industrial control and demonstrating their application to an energy-related use case.

The specific Sustainable Development Goals mentioned have the following names:

SDG 7 Ensure access to affordable, reliable, sustainable and modern energy for all

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

SGD	Target	Contribution	Performance Indicators and Metrics
7	7.3	The energy-related use case demonstrates how modular CPS architectures can support real-time energy monitoring, contributing to more efficient control decisions.	Monitoring granularity/frequency; computational overhead relative to monitoring benefit
9	9.4	Modular, reconfigurable automation architectures (CFBs on IEC 61499) support incremental upgrades of industrial control systems, reducing the need for full infrastructure replacement.	Degree of component reuse; reconfiguration effort required to adapt the system
	9.5	Contributes to technological and scientific understanding of CFB deployment and execution on modern RTEs.	Research contribution

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

Acknowledgements

Esta tese marca o fim de um longo percurso e como tal tenho muitas pessoas às quais quero agradecer, inicialmente vou falar sobre as pessoas que me acompanharam e orientaram ao longo deste projeto e posteriormente falarei daqueles que foram essenciais na minha vida e percurso académico.

Quero começar por agradecer ao meu orientador o professor Rui Pinto, agradeço pela forma como me orientou e pela autonomia que me deu ao longo deste projeto. Agradeço também a presença constante nos momentos em que ia pedir orientação e conselhos sobre os problemas que foram aparecendo. E por último agradeço a disponibilidade que sempre teve ao longo do projeto, tanto nos momentos em que tivemos de remarcar reuniões ou quando eu precisava que verifica-se algo rapidamente porque apareceu um erro inesperado, o professor arranjou sempre disponibilidade para me apoiar e orientar na resolução destes pontos que foram aparecendo. Muito Obrigado.

Gostaria também de agradecer aos engenheiros João Reis e Eliseu Pereira, pela ajuda, discussão e conselhos técnicos. Foram importantes guias nas questões técnica relacionadas ao DINASORE e as alterações que foram necessárias para o projeto. As sessões de apresentação sobre os projetos de tese do DIGI2 tiveram uma relevante importância para o desenvolvimento do meu projeto. Os conteúdos foram clarificantes e orientadores de estratégias a desenvolver. Além de noções em que fiquei mais esclarecido e elucidado.

Por último, relativamente a esta tese, quero agradecer ao estudante Pedro Mota, pelo apoio prestado ao longo do projeto em trouble shooting e programas que me ajudaram a resolver problemas ao longo da tese.

Passando para os agradecimentos a pessoas que são muito queridas e que acompanharam-me ao longo da Licenciatura e do Mestrado, quero começar por agradecer aos meus pais Luís Magalhães e Carla Mendes. Sou vos muito grato por tudo o que fizeram por mim ao longo da minha vida. Não teria chegado tão longe se não fosse com a vossa ajuda. Estou eternamente grato aos meus maiores e eternos apoiantes que sempre me amaram e orientaram desde o primeiro dia da escola até ao agora ao final. Muito obrigado por terem sido estes grandes marcos na minha vida, até nos dias em estavam uns chatos. Amo-vos muito.

Continuando quero também agradecer a uma baixinha muito especial, a minha irmã Maria. És uma chata, mas és a minha chata, sei que posso contar contigo até ao fim. E parece que o fim desta etapa está a chegar, vais ter o teu último ano de faculdade e como alguém que acaba de viver o seu último digo-te, aproveita muito, ele passa a correr. Sempre foste um grande apoio meu ao longo desta caminhada e não teria chegado aqui sem ti. Muito Obrigado.

Por último quero agradecer a um grupo de pessoas que me acompanharam nestes anos de faculdade. Alguns já acabaram, outros ainda estão a acabar, mas todos estão comigo hoje. E como não podia ser doutra forma dedico-vos este poema:

Ao Bando

No jardim da faculdade nos plantámos, um bando de mal formados, sem receio, entre aulas e cafés que atrasámos, fomos crescendo, cada um ao seu meio.

Marquise a abrir a estrada com estilo, Gemas, algemas presas em sigilo, Cavalete, uterino no seu jeito, Reforço, colher em riste, sempre direito.

Pisca sempre a piscar o olho à sorte, Xanax a acalmar quando é preciso, Orgânica, fadista de improviso, o bando fechava-se ali, sem corte.

Fica aqui a palavra de apreço, a todos que este jardim semeou, pelo riso, pela festa, pelo abraço, o bando e as amizades, tudo que ficou.

<Gonçalo Magalhães>

"Foi bom, enquanto durou, mas hoje acabou, tudo passou"

DJ Ademar e Boy Teddy

Contents

1 Introduction **1**
- 1.1 Problem Statement . . . 2
- 1.2 Objectives . . . 2
- 1.3 Document Structure . . . 3

2 State of the Art **5**
- 2.1 Function Blocks in CPS . . . 5
- 2.2 Composite Function Blocks and Hierarchical System Modeling . . . 8
- 2.3 Run-Time Enviroments for IEC 61499 . . . 10
- 2.4 Integrated Development Enviroments for IEC 61499 . . . 11
- 2.5 Speculative Computing in Resource-Constrained Systems . . . 14
- 2.6 Research Gap . . . 17

3 Methodology **19**
- 3.1 Target RTE and IDE . . . 20
  - 3.1.1 DINASORE RTE . . . 20
  - 3.1.2 4diac IDE . . . 21
- 3.2 Distributed pipeline architecture . . . 21
- 3.3 Pipeline Configuration . . . 23
- 3.4 Memoization-based Caching Mechanism . . . 24
- 3.5 System Monitoring and Metrics Collection . . . 26

4 Implementation **27**
- 4.1 Hardware and Software Setup . . . 27
  - 4.1.1 Hardware Infrastructure . . . 27
  - 4.1.2 Software Stack . . . 28
  - 4.1.3 DINASORE Modifications . . . 28
- 4.2 Atomic FB Design . . . 35
  - 4.2.1 SENSOR_SIMULATOR . . . 35
  - 4.2.2 HVAC_ENERGY_CALC . . . 35
  - 4.2.3 MQTT_PUBLISHER, MQTT_SUBSCRIBER and INFLUX_DB_2 . . . 36
- 4.3 Pipeline CFB . . . 37
- 4.4 Deployment with 4diac and DINASORE . . . 39
- 4.5 Monitoring Infrastructure . . . 40
  - 4.5.1 System Metrics Collection and Visualization . . . 40

5 Experimental Evaluation **41**
5.1 Experimental Setup . . . 41
5.2 Evaluation Objectives . . . 41
5.3 Test Scenarios . . . 42
5.3.1 Single-sensor baseline . . . 42
5.3.2 Single-sensor full_cfb . . . 43
5.3.3 Multi-sensor baseline . . . 45
5.3.4 Multi-sensor full_cfb . . . 46

6 Discussion **49**
6.1 Impact of CFBs on Computational Overhead . . . 49
6.2 Effectiveness of Memoization-Based Speculative Reuse . . . 51
6.3 CPU, Memory and Observability Trade-offs . . . 53
6.3.1 Encapsulation vs Observability . . . 53
6.3.2 CPU vs Memory . . . 53
6.3.3 Determinism vs Throughput . . . 53
6.3.4 Modularity, Reuse and Scalability . . . 54
6.4 Limitations and Threats to Validity . . . 54
6.5 Design Guidelines for CFB-Based CPS . . . 56
6.6 Validation of the Research Hypothesis . . . 57

7 Conclusion **59**
7.1 Limitations of the Proposed Solution . . . 60
7.2 Future Work . . . 60

References **62**

A Appendix **66**

List of Figures

2.1 Structure of an IEC 61499 FB [21]. 5
2.2 Structure of an IEC 61499 ECC [15]. 6
2.3 Structure of an IEC 61499 CFB compared to a FB [22]. 8

3.1 Diagram of the CPPS. 22
3.2 Diagram of the distributed architecture. 22
3.3 Diagram exemplifying both configurations. 24
3.4 Flowchart of the SPECULATIVE-GATE. 25

4.1 Complete diagram of the distributed architecture. 28
4.2 UML class diagram of *FBInterface* hierarchy. 30
4.3 Execution flow of the original DINASORE runtime engine. 31
4.4 Execution flow of the extended DINASORE runtime engine, with separate branches for *AtomicFB* and *CompositeFB* execution. 31
4.5 Idempotency guard in `create_virtualized_fb()`, preventing duplicate OPC UA node creation on repeated deployment requests. 33
4.6 Deployment routing flow in the extended `Manager` class, dispatching CREATE requests to `AtomicFB` or `CompositeFB` handlers. 34
4.7 *SENSOR_SIMULATOR* FB. 35
4.8 *HVAC_ENERGY_CALC* FB. 36
4.9 *MQTT_PUBLISHER_2* FB. 36
4.10 *MQTT_SUBSCRIBER_2* FB. 36
4.11 *INFLUX_DB_2 FB*. 37
4.12 *PIPELINE_P1* CFB. 37
4.13 Internal FB network of *PIPELINE_P1* without *SPECULATIVE_GATE*. 38
4.14 PIPELINE_PC CFB external interface, aggregating the configuration parameters of the internal MQTT_SUBSCRIBER_2 and INFLUX_DB_2 FBs into a single deployable unit for the management node. 38
4.15 Internal FB network of PIPELINE_PC, showing two MQTT_SUBSCRIBER_2 instances subscribing to the edge node topics and forwarding data to a shared INFLUX_DB_2 FB for time-series storage. 39
4.16 Data flow of the monitoring infrastructure, from metric collection on each node to storage in InfluxDB and visualization in Grafana. 40

5.1 CPU usage during the single-baseline test. 43
5.2 Memory usage during the single-baseline test. 43
5.3 CPU usage during the (S2) test. 44
5.4 Memory usage during the (S2) test. 45
5.5 CPU usage during the multi-baseline test. 46

5.6 Memory usage during the multi-baseline test. 46
5.7 CPU usage during the multi-full_cfb test. 47
5.8 Memory usage during the multi-full_cfb test. 47

List of Tables

2.1 Summary of related work and research gaps addressed by this work. Columns indicate whether each contribution addresses CFB execution overhead characterization (CFB overhead), memoization-based speculative reuse within IEC 61499 (Memo.), and empirical deployment design Guidelines. ✓ = addressed; × = not addressed. 17

5.1 (S1) results. 43
5.2 (S2) results. 44
5.3 (S3) results. 46
5.4 (S4) results. 47

List of Acronyms

CFB	Composite Function Block
CPU	Central Processing Unit
CPS	Cyber-Physical Systems
CPPS	Cyber-Physical Production Systems
DINASORE	Dynamic Intelligent Architecture for Software Modular Reconfiguration
DSL	Domain-Specific Language
ECC	Execution Control Chart
FB	Function Block
FBME	Function Block Modeling Environment
FBSRT	Function Block Service Runtime
IDE	Integrated Development Environment
IEC	International Electrotechnical Commission
IoT	Internet of Things
MQTT	Message Queuing Telemetry Transport
OPC UA	Open Platform Communications Unified Architecture
RAM	Random Access Memory
RTE	Run-Time Environment
SCP	Security Copy Protocol
TCP	Transmission Control Protocol
TLS	Thread-Level Speculation
XML	Extensive Markup Language

Chapter 1

Introduction

The increasing adoption of Industry 4.0 principles has fundamentally altered the design and operation of industrial automation systems [1]. Modern production environments are no longer composed of isolated control units but rather of interconnected Cyber-Physical Systems (CPS) that tightly integrate sensing, actuation and communication [2] . This integration enables real-time monitoring, decentralized decision making and adaptive behavior, but it also results in a marked increase in system complexity [3].

As industrial systems evolve towards distributed and intelligent architectures, traditional centralized control approaches based on fixed logic and monolithic software structures become insufficient [4], [5]. These legacy solutions face limitations in terms of scalability, flexibility and maintainability, particularly when confronted with dynamic reconfiguration requirements and heterogeneous device ecosystems [6]. Consequently, new modeling and execution paradigms are required to effectively manage the complexity inherent to modern CPS.

The International Electrotechnical Commission (IEC) 61499 standard addresses these challenges by providing an event-driven model for distributed control systems [4], [7], [8], centered on the concept of Function Block (FB) as modular software components. By separating control logic from execution platforms and enabling explicit event handling, IEC 61499 supports the development of portable and reconfigurable automation applications [9]. Within this standard, Composite Function Block (CFB) offer an additional abstraction layer, allowing complex system behavior to be modeled through hierarchical composition [7], [10], thus aligning naturally with the structure of Cyber-Physical Production Systems (CPPS).

However, the practical realization of IEC 61499-based CPS critically depends on the capabilities of the underlying Run-Time Environment (RTE) [11]. Whether the theoretical benefits of CFBS, hierarchical composition, encapsulation and modularity, translate effectively into operational behavior is determined by how the RTE models and schedules these constructs. This relationship between the IEC 61499 abstraction layer and runtime realization introduces tradeoffs in terms of computational overhead, deployment complexity and system observability that are not yet well characterized, particularly for CPPS with multiple interconnected components.

Against this backdrop, this thesis examines how CFBs can be practically employed within a

modern RTE for distributed industrial control, evaluating how hierarchical composition performs under realistic CPPS conditions. The work seeks to contribute to the broader understanding of when and how IEC 61499 modular abstraction deliver their intended benefits in practice and to identify the trade-offs that engineers must consider when adopting CFB-based designs in distributed industrial systems.

A further consequence of CFB encapsulation. not previously explored, is that the CFB boundary creates a natural site foe embedding optimization mechanisms. By hiding an entire pipeline behind a well-defined interface, a CFB allows techniques such as memoization-based speculative reuse to be introduced transparently, without modifying the enclosed FBs or the external application logic. This thesis exploits that property to investigate wether CFB encapsulation can simultaneously improve modularity and reduce computational load on resource-constrained edge hardware.

1.1 Problem Statement

CFBs, as defined by IEC 61499, offer a conceptually elegant solution to the complexity of distributed CPS: they allow control logic to be organized hierarchically, encapsulating lower-level behavior into reusable, modular units. In principle, this should make complex systems easier to design, understand and reconfigure. However, whether this theoretical benefit translates into practice depends on the RTE responsible for executing these models, and this relationship is not yet well understood.

Most existing RTE's were designed with a focus on deterministic, real-time execution of relatively simple, FB control logic [12], [13], [14].While the literature has addressed execution semantics [11], [13], [15], portability [9] and formal verification of IEC 61499 systems [16], [17], the practical trade-offs introduced specifically by CFB hierarchical composition in modern RTEs, namely computational overhead, deployment complexity and system observability, remain largely uncharacterized. Without this understanding, engineers lack the guidance needed to decide when and how CFBs should be adopted in real distributed control applications.

A related and unexplored question is wether the encapsulation boundary introduced by a CFB can itself serve as an optimization point. By hiding a computationally intensive pipeline behind a well-defined interface, a CFB provides a contained context in which a caching mechanism can be introduced at specific internal stages, reducing computational load without exposing that optimization to the external application. This positions CFB encapsulation not merely as a structural tool, but as an enabler of deployment-transparent optimization.

1.2 Objectives

The central hypothesis driving this work is:

H1: Hierarchical composition through CFBs can improve modularity and deployment flexibility in IEC 61499 applications without introducing significant runtime overhead supported by a modern Python-based RTE.

To validate **H1**, three specific research questions are addressed:

RQ1 What runtime overhead is introduced by CFBs when compared with equivalent flat FB networks?

RQ2 How do CFBs affect deployment characteristics, modularity and maintainability of distributed IEC 61499 applications?

RQ3 Can memoization-based speculative reuse compensate for computational costs in CFB-based edge deployments?

To address these questions, this thesis pursues the following objectives:

1. Explore modeling techniques for complex CPS using IEC 61499 CFBs, examining how hierarchical composition can be applied to represent distributed control behavior.
2. Evaluate the capabilities of Dynamic Intelligent Architecture for Software Modular Reconfiguration (DINASORE) as a RTE for deploying and executing CFB-based models, in the context of distributed automation systems.
3. Characterize the trade-offs introduced by hierarchical composition, contributing to a broader understanding of how modularity and abstraction interact with the practical constraints of distributed, real-time execution.
4. Propose improvements and design guidelines for the effective use of CFBs in DINASORE-based distributed industrial systems.

1.3 Document Structure

This document is organized into seven chapters.

Chapter 2 State of the Art — reviews the relevant literature on FBs in CPS, CFBs and hierarchical system modeling, RTEs and Integrated Development Environment (IDE)s for IEC 61499.

Chapter 3 Proposed Solution — presents the approach adopted to explore the use of CFBs within DINASORE, including the architecture and strategies considered to model and evaluate the system under study.

Chapter 4 Implementation — describes how the proposed solution was implemented, including the hardware and software setup and the design of the FBs used.

Chapter 5 Experimental Evaluation — presents the experimental protocol followed and the results obtained from evaluating the implemented solution.

Chapter 6 Discussion — analyzes and interprets the results obtained, discussing the trade-offs observed, the limitations of the study and the design guidelines that emerge from it.

Chapter 7 Conclusion — summarizes the contributions of this thesis and outlines directions for future work.

Chapter 2

State of the Art

2.1 Function Blocks in CPS

The IEC 61499 standard establishes FBs as the fundamental abstraction for the design and implementation of distributed industrial automation systems [4], [8], [18], [19], [20]. An IEC 61499 application is defined as a network of interconnected FB instances that collectively implement system behavior across multiple devices and execution contexts [9], [10]. This paradigm explicitly targets distributed automation and CPSs, addressing limitations of centralized and scan-based control architectures [4], [6], [20]. Figure 2.1 illustrates the structure of an IEC 61499 FB, showing the separation between event and data interfaces.

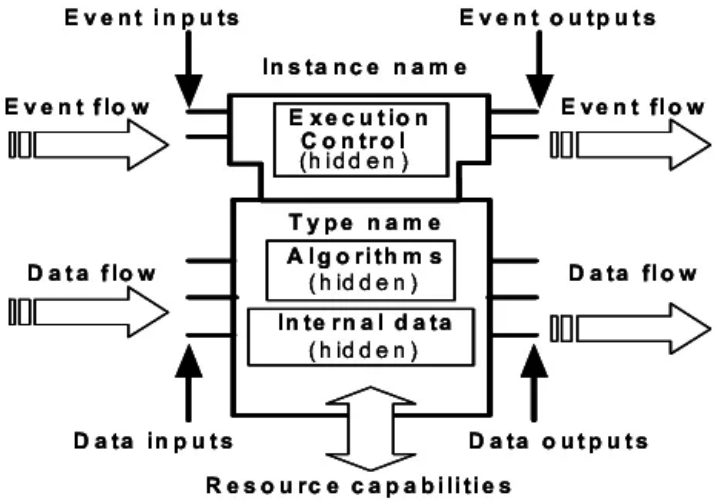


Figure 2.1: Structure of an IEC 61499 FB [21].

An IEC 61499 FB encapsulates both interface and behavior. The interface consists of event inputs, event outputs, data inputs and data outputs, providing a standardized interaction contract between components [4], [9]. Events are responsible for controlling orders, while data connections transport values independently of control flow [9], [18]. This strict separation of event and data

domains is a defining characteristic of IEC 61499 and enables explicit modeling of causality and synchronization [4], [9].

The internal behavior of a basic FB is specified using an Execution Control Chart (ECC) [9]. The ECC is an executable state machine in which transitions are triggered by incoming events and guarded by Boolean conditions [9]. Each ECC state may be associated with one or more algorithms and state transitions may emit output events [9]. This structure provides a formal and analyzable execution model that supports verification, simulation and runtime monitoring [9], [16]. Figure 2.2 shows th structure of an IEC 61499 ECC, depicting the state machine that governs FB execution.

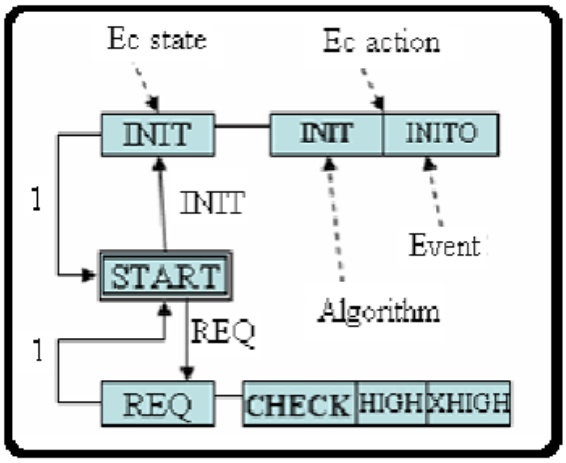


Figure 2.2: Structure of an IEC 61499 ECC [15].

Beyond its structural definition, the IEC 61499 FB model introduces an execution paradigm that distinguishes it fundamentally from traditional control approaches.

Unlike IEC 61131-3, which relies on cyclic scanning of logic, IEC 61499 adopts an event driven execution model [5], [6], [7], [18]. Execution occurs only when an event arrives at an FB, eliminating unnecessary computation and enabling fine-grained control over execution order [9]. This approach is particularly suited to distributed systems, where events may originate from sensors, communication interfaces or other distributed components [4], [9].

In industrial automation practice, FBs are used to encapsulate control algorithms, communication services, data preprocessing and coordination logic [18]. The standard allows algorithms within FBs to be written in multiple programming languages, including IEC 61131-3 languages and general-purpose languages such as C++ and Python [12]. This flexibility supports the integration of operational technology and information technology within a unified system model [12].

This execution flexibility is especially relevant when FBs are applied within CPSs, where control software must respond to asynchronous physical stimuli rather than fixed scan cycles.

Event-driven FB execution is particularly relevant for CPSs, where software behavior must be synchronized with asynchronous physical processes and network communication [2], [4], [6]. By reacting to events rather than relying on fixed scan cycles, IEC 61499 supports responsive and scalable control architectures suitable for dynamic, distributed environments [4], [9].

In CPPS, control software is tightly coupled with physical processes through continuous sensing, actuation and communication [6], [16]. IEC 61499 FBs provide a component-based abstraction that explicitly supports this coupling by modeling software behavior as reactive components driven by events originating from the physical environment or from distributed communication channels [6], [18].

Sensors, actuators and communication interfaces in CPS generate asynchronous stimuli that cannot be efficiently handled using strictly cyclic execution models [4], [6]. The event-driven nature of IEC 61499 FBs enables direct representation of such stimuli as events that trigger control logic only when relevant changes occur [6], [9]. This allows control behavior to be synchronized with physical phenomena such as state transitions, threshold crossings or communication updates, rather than being constrained by fixed scan periods [4], [6].

FBs also facilitate the integration of heterogeneous cyber and physical components within a unified system model [6], [18]. In IEC 61499-based CPS, FBs may encapsulate real-time control algorithms, signal processing, communication protocols and higher-level decision logic within standardized interfaces [12], [18]. This supports the coexistence of operational technology and information technology components within the same application while preserving modularity and separation of concerns [10], [12].

These properties also extend naturally to distributed deployment and formal analysis, two further dimension in with IEC 61499 FBs offer concrete advantages.

From a system architecture perspective, FBs act as cyber representations of physical and logical entities [6], [16]. A single physical device or subsystem may be represented by one or more FBs that expose events and data corresponding to its observable behavior and controllable actions [6], [9]. This mapping enables explicit modeling of interactions between physical processes and distributed control logic, which is a defining characteristic of CPSs [6], [16].

Furthermore, the distribution capabilities of IEC 61499 allow FBs to be deployed across multiple devices and network nodes, reflecting the decentralized nature of CPS [9], [10]. Event and data connections may represent either local interactions or remote communication, abstracting underlying transport mechanisms [9], [18]. This abstraction enables scalable CPS architectures in which control and coordination logic can be dynamically distributed across edge devices, controllers and gateways [4], [18], [20].

The explicit execution semantics of IEC 61499 FBs also support formal analysis and verification of CPS behavior [16], [17]. Since FB execution is governed by well-defined event-driven state machines, system behavior can be analyzed through execution traces obtained from simulation, runtime monitoring or formal verification tools [16], [17]. This capability is particularly relevant for CPS, where errors in control logic may lead to unsafe physical behavior and high economic cost [4], [17].

2.2 Composite Function Blocks and Hierarchical System Modeling

As industrial automation systems increase in size and complexity, flat FB networks become difficult to manage, understand and maintain [10]. The IEC 61499 standard addresses this challenge through CFBs, which enable hierarchical modeling by grouping multiple FBs into a single reusable unit [9], [10]. Figure 2.3 compares the structure of a CFB with that of a FB, highlighting the encapsulation of an internal FB network.

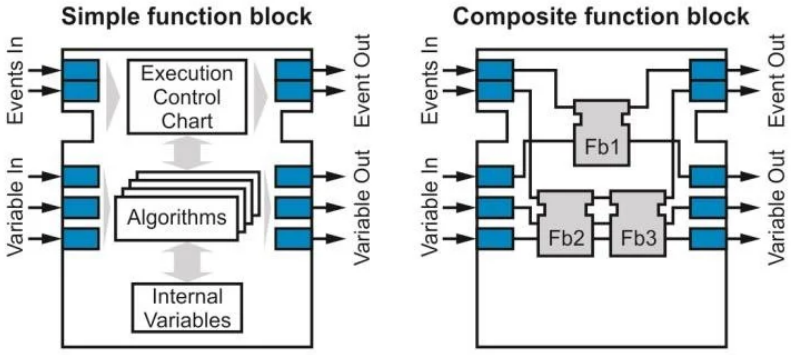


Figure 2.3: Structure of an IEC 61499 CFB compared to a FB [22].

A CFB internally contains an FB network with event and data connections, while externally exposing an interface identical in nature to that of a basic FB [9]. This allows CFBs to be treated as black-box components at higher levels of abstraction [9], [10]. Internal implementation details are hidden, enabling designers to reason about system behavior at multiple levels without being overwhelmed by low-level complexity [10].

This black-box abstraction is not merely structural, it enables an entire approach to hierarchical system design that scales naturally to the complexity of industrial automation.

Hierarchical modeling with CFBs is essential for managing complex industrial systems such as production lines, robotic cells and industrial edge platforms [9], [10]. Lower-level FBs implement elementary functionality, while higher-level CFBs represent coordinated subsystems, communication patterns or functional modules [9], [10]. This hierarchical decomposition mirrors physical system structure and supports systematic engineering practices [6], [10].

One of the primary advantages of CFBs is modularity. By encapsulating functionality into well-defined units, CFBs enable independent development, testing and reuse [10]. A CFB developed for one application can be instantiated multiple times or reused in different projects with minimal modification if interface contracts are preserved [10].

CFBs also enhance scalability. As systems evolve, new functionality can be introduced by adding or extending CFBs rather than modifying existing logic [6], [10]. This property is particularly valuable in industrial contexts where systems must adapt to changing requirements, equipment upgrades or reconfiguration of production processes [6], [10].

These modularity and scalability benefits become even more significant when CFBs are applied in the context of CPSs, where software must interact tightly with distributed physical processes.

In CPS, complexity arises not only from the control logic itself, but also from the tight interaction between distributed software components and physical processes operating at different temporal and spatial scales [6], [16]. CFBs provide an abstraction mechanism that aligns naturally with CPS design by allowing physical subsystems and cyber coordination logic to be represented as hierarchical software components with clearly defined interfaces [6], [9], [10].

CFBs enable the encapsulation of complete CPPS such as motion control units, production cells or communication gateways, within a single functional entity [9], [10]. Internally, CFBs may integrate FBs responsible for sensing, actuation, signal conditioning, communication and supervisory logic [6], [9]. Externally, the CFB exposes only the events and data required for interaction with other subsystems, reflecting the physical and functional boundaries of the CPS architecture [9], [10].

Hierarchical modeling with CFBs supports the decomposition of CPS into interacting layers of abstraction [9], [10]. Lower-level FBs typically operate close to the physical process, handling real-time signal processing or device interaction, while higher-level CFBs implement coordination, sequencing and decision-making logic [6], [12]. This separation allows CPS designers to distinguish between time-critical control functions and higher-level cyber coordination without violating the unified execution model of IEC 61499 [6], [10].

In distributed CPS, where subsystems may be deployed across multiple edge devices or controllers, CFBs also facilitate scalable system composition[9], [10]. A CFB may encapsulate functionality that is internally distributed across several resources, while appearing as a single logical unit at the system level[9], [10]. This abstraction reduces the cognitive load on system designers and supports incremental system evolution as CPS grows and functional scope[6], [9].

Despite these advantages, the adoption of CFBs in CPS deployments is not without challenges, particularly when physical interaction and runtime observability are considered.

However, the use of CFBs in CPS also introduces challenges that are amplified by physical interaction [16], [17]. Limited observability of internal FB execution can hinder diagnosis of faults that manifest unexpected physical behavior [17], [23]. Furthermore, event propagation across hierarchical CFB boundaries may introduce execution delays or nondeterministic interactions if not effectively managed by the RTE [16], [17]. These challenges highlight the importance of runtime support and tool assistance for monitoring and analyzing hierarchical execution in cyber-physical deployments [17], [23].

The use of CFBs in IEC 61499 has been extensively studied in the context of hierarchical modeling, modularity and system scalability [6], [10], [24], [25]. CFBs have been proposed as the primary abstraction mechanism for managing complexity in large-scale distributed automation systems, enabling encapsulation, reuse and separation of concerns across deployment boundaries [4], [10]. In multi-agent and distributed CPS architectures, CFBs have been used to represent autonomous subsystems with well-defined interfaces [6].

However, in all existing work, CFBs are treated exclusively as structural and organizational abstractions. Their role is limited to encapsulation and deployment of FB networks. The computational behavior introduced by hierarchical composition, and the overhead that CFB boundaries impose at runtime, has not been systematically characterized in the literature.

2.3 Run-Time Enviroments for IEC 61499

The execution of IEC 61499 applications is realized through RTEs [9], [20], [26]. An RTE is responsible for executing FB networks, managing event queues, scheduling algorithms, updating data values and handling communication between distributed components [26]. The behavior of an IEC 61499 system is therefore strongly influenced by the characteristics of the underlying RTE [9], [26].

Execution semantics are a central concern in IEC 61499 RTE design [9], [26]. The standard itself does not mandate a single execution model, allowing different RTEs to implement different scheduling and concurrency strategies [9], [26]. As a result, research has focused on defining deterministic execution semantics to ensure predictable system behavior in distributed environments [12], [17].

Several distinct strategies have emerged to address this challenge, each reflecting priorities in terms of determinism, concurrency and real-time performance.

Several approaches propose prioritized and concurrent deterministic execution models [12], [18]. These approaches control the order in which events are processed and FBs are executed, reducing nondeterminism caused by concurrent event arrivals [12], [13]. Such semantics are particularly relevant for industrial edge applications, where multiple services and control functions coexist on shared hardware platforms [12], [13].

Edge-oriented RTEs focus on flexibility rather than strict real-time guarantees [12], [13]. These RTEs support deployment of heterogeneous devices that combine control, communication, data acquisition and data processing [13], [18]. In such environments, IEC 61499 is primarily used for coordination and orchestration rather than hard real-time control, which remains delegated to lower-level controllers [12], [13].

These execution models are realized in practice through set of established IEC 61499 RTEs, each embodying different trade-offs between determinism, flexibility and platform support.

One widely used IEC 61499 runtime is FORTE [27], which supports event-driven execution, distributed deployment and execution of both basic and CFBs [9], [26]. In FORTE, CFBs are executed by resolving their internal FB networks within the runtime while preserving their external interfaces [11], [24]. This enables hierarchical modeling at the application level while relying on a uniform execution mechanism internally [10], [11]. Composite structures are therefore supported without introducing specialized execution primitives, although internal execution visibility is limited at runtime [11], [26].

Another execution environment is the Function Block Service Runtime (FBSRT), which is explicitly designed to support industrial edge applications [12], [13]. This runtime executes IEC

61499 applications that integrate control logic, communication services and data processing tasks on edge devices [13], [18]. FBSRT supports CFBs as containers for communication segments, protocol handling and logic coordination [28]. CFBs are used to encapsulate complex communication functionality while exposing simplified interfaces to the application layer, reducing complex design in large systems [10], [28].

Several IEC 61499 RTEs have been developed with a focus on deterministic and prioritized execution semantics, targeting industrial edge and CPS [12], [13]. These RTEs implement explicit rules for event ordering, concurrency and scheduling to ensure predictable execution behavior [12], [13]. Deterministic semantics are essential for the correct execution of CFBs, as nested FB networks may generate concurrent events who's ordering directly influences system behavior [12], [26]. Without controlled scheduling, hierarchical execution may lead to nondeterministic outcomes [17], [26]. Lightweight, Python-based IEC 61499 RTEs targeting rapid prototyping on resource-constrained single-board computers, and their suitability as platforms for investigating higher-level optimization strategies within CFBs, remain largely unaddressed in the literature [29].

A further dimension in which RTEs vary is their integration with service-oriented communication standards, particularly Open Platform Communications Unified Architecture (OPC UA), which extends IEC 61499 execution broader industrial information architectures [30].

RTEs integrating OPC UA further extend IEC 61499 execution by embedding FB applications within service-oriented and information-centric architectures [18], [28], [30], [31]. OPC UA enabled RTEs expose FB instances and their interfaces through standardized information models allowing runtime creation, monitoring and interaction with FBs [18]. CFBs can be represented as structured information entities, enabling hierarchical subsystems to be monitored and reconfigured at runtime while maintaining encapsulation of internal behavior [18].

2.4 Integrated Development Enviroments for IEC 61499

IDEs are a key enabling element for the engineering of IEC 61499-based automation systems, as they provide the means to model, structure and manage FB applications throughout their life cycle [11], [32]. In contrast to traditional programming environments, IEC 61499 IDEs must support event-driven execution models, hierarchical system decomposition through CFBs and configuration of distributed deployments across multiple devices and resources [9], [32].

An IEC 61499 IDE typically supports graphical modeling of FBs and CFBs, definition of event and data connections, system configuration across multiple devices and integration with one or more RTEs [9], [32]. These capabilities enable engineers to manage the structural and behavioral complexity inherent to distributed control systems while maintaining a consistent system model throughout the development life cycle [10], [32].

Among these capabilities, hierarchical modeling and system configuration stand out as foundational elements that determine how effectively engineers can manage structural and deployment complexity.

IEC 61499 IDEs provide a visual and structural abstraction layer that allows developers to construct applications using networks of FBs and CFBs [9], [32]. Graphical modeling aligns with established industrial engineering practices and supports comprehension of control logic, communication patterns and system architecture [10], [32]. However, as application size increases, flat FB networks rapidly become difficult to manage [18], [32].

To address this issue, modern IDEs provide explicit support for hierarchical modeling through CFBs [10], [32]. CFBs allow subsystems to be encapsulated and reused, reducing visual complexity and enabling separation of concerns [9], [10]. IDE features such as hierarchical navigation, interface abstraction and modular libraries are therefore essential for scaling IEC 61499-based systems to industrial-size applications [9], [32].

In CPSs where control logic must interact with physical processes, communication infrastructure and higher-level coordination mechanisms, IDEs also support system-level configuration [9], [32]. This includes mapping FB instances to devices and resources, defining network segments and configuring communication interfaces [9], [10]. Such features allow the same functional model to be deployed across different physical architectures [9], [32].

Beyond modeling, IEC 61499 IDEs support deployment of applications to target RTEs [9], [32]. This includes generation of deployment configurations, resource assignments and communication mappings [9], [10]. IDEs act as the central coordination point between design-time abstractions and runtime execution platforms [11], [32].

Beyond the initial design and deployment phase, IDEs also play a central role i the ongoing maintenance, evolution and debugging of IEC 61499 applications across their operational lifetime.

Maintenance and evolution of IEC 61499 applications are also strongly influenced by IDE support [10], [32]. As systems evolve over time, IDEs facilitate modification of FB networks, replacement or extension of CFBs and re-deployment of updated applications [9], [32]. This capability is particularly relevant in CPSs, where continuous adaptation and reconfiguration are common [6], [10].

Support for monitoring and debugging further extends the role of IDEs [23], [32]. Visualization of execution states, event flows and data values allows developers to identify faults and understand system behavior [23], [26]. Without adequate tool support, debugging distributed and hierarchical IEC 61499 applications becomes impractical [23], [32].

These requirements have shaped the design of several notable IEC 61499 IDEs, each addressing different aspects of modeling, execution transparency and integration with RTEs.

The Function Block Modeling Environment (FBME) is a model-centric IEC 61499 IDE designed to support advanced modeling, analysis and verification of FB applications [23]. FBME provides comprehensive support for FBs and CFBs, hierarchical modeling and explicit representation of execution semantics [23].

FBME distinguishes itself from conventional IEC 61499 IDEs by integrating a semantic execution framework [11], [23]. This framework allows IEC 61499 models to be executed within the IDE for purposes of simulation, analysis and verification [11], [26]. Execution traces captured

by FBME include detailed information about event selection, ECC state transitions, algorithm execution and data updates [23].

Because FBME can execute IEC 61499 models, it may appear to function as a RTE [23]. However, this execution capability is intended for development-time use, not for deployment on industrial devices [23]. FBME does not provide the scheduling guarantees, hardware integration or communication infrastructure required of an industrial RTE [10], [11]. Instead, it serves as a design-time execution engine that enables deep inspection of system behavior [23], [26].

This dual role explains why FBME is sometimes perceived as both an IDE and an RTE [23]. In practice, FBME should be understood as an IDE that incorporates execution semantics to support validation and verification activities [23], [26]. Deployment of IEC 61499 applications developed in FBME is performed on external RTEs, such as industrial or edge-oriented IEC 61499 RTEs [9], [28].

In addition to FBME, several IDEs have been developed to support IEC 61499-based engineering, each emphasizing different aspects of modeling, scalability, usability and integration with execution platforms [9], [32]. These environments reflect diverse design goals, ranging from industrial deployment to experimental modeling and domain-specific abstraction [9], [23].

One of the most widely adopted IEC 61499 IDEs is the Eclipse 4diac IDE, which provides comprehensive support for graphical modeling of FBs and CFBs, system configuration and deployment to IEC 61499-compliant RTEs [7], [9], [32]. The 4diac IDE supports hierarchical modeling through CFBs, library management and device/resource configuration, making it suitable for engineering distributed automation systems [10], [32]. Its tight integration with the FORTE RTE enables a streamlined workflow from design to deployment, although execution introspection at runtime remains limited compared to model-centric environments [23], [32].

While 4diac supports deployment and execution, its primary focus remains on design-time modeling and system configuration [9], [32]. Runtime introspection and hierarchical execution tracing are limited compared to model-centric environments, which can complicate debugging of deeply nested CFBs in large-scale systems [23], [32].

A growing body has evaluated more advanced IDE capabilities and their specific influence on the practical adoption and effective use of CFBs in larger systems.

Usability and scalability of visual IEC 61499 IDEs have been explicitly evaluated in the context of large-scale automation software [32]. Visual programming environments support intuitive modeling for small and medium-sized systems; however, as system size increases, navigation, readability and cognitive load become significant challenges [10], [32]. Hierarchical structuring through CFBs is identified as a necessary mechanism to mitigate visual complexity, but its effectiveness depends strongly on IDE features such as hierarchical navigation, modular libraries and abstraction support [9], [32]. IDEs lacking advanced hierarchical tooling hinder effective system comprehension and maintenance in large projects [10], [32].

To address limitations of purely graphical approaches, Domain-Specific Language (DSL), based IEC 61499 development tools have been proposed [32], [33]. These tools introduce textual or semi-textual representations tailored to specific application domains, allowing engineers to

describe control logic, constraints and architectural patterns at a higher level of abstraction [18], [33]. DSL-based IDEs enable automatic generation of IEC 61499 FBs and CFBs from domain-level specifications, reducing manual modeling effort and minimizing design errors [18], [33]. Such approaches are particularly effective for repetitive or pattern-based system structures [18].

Model-based engineering environments further extend IEC 61499 IDE capabilities by supporting transformation from higher-level system models to executable FB applications [18], [33]. Tools enabling transformation from SysML system design models to IEC 61499 FB networks facilitate early integration of control software design with system-level architectural modeling [18]. This approach supports traceability between requirements, system architecture and executable control logic, which is especially relevant for CPSs with strong coupling between physical structure and software behavior [10], [18].

Several IEC 61499 IDEs incorporate formal modeling and verification capabilities to support rigorous analysis of execution behavior [11], [17], [23]. These environments provide explicit representations of execution traces, event causality and state transitions, enabling formal reasoning about system behavior [23], [26].

Formal execution models are especially valuable for systems composed of deeply nested CFBs, where emergent behavior may not be evident from static models alone [23], [26]. IDEs supporting formal analysis enable early detection of logical errors, unintended concurrency and unsafe execution patterns before deployment to RTEs [17], [23].

The effective use of CFBs is strongly influenced by IDE capabilities [23], [32]. While CFBs provide a powerful abstraction mechanism, insufficient tool support can obscure internal behavior and hinder debugging and maintenance [10], [32]. IDEs that provide limited visibility into hierarchical execution discourage extensive use of CFBs in practice [23], [32].

Conversely, IDEs that support hierarchical modeling, execution tracing, model transformation and formal analysis enable CFBs to be used as first-class architectural elements [17], [18], [23]. In such environments, CFBs become a practical and scalable mechanism for structuring complex CPPS rather than a purely conceptual abstraction [10], [32].

While IDE tooling addresses the engineering complexity of developing and deploying CFB-based applications, it does not address the computational overhead that hierarchical CFB execution may introduce on the resource-constrained edge devices where these applications are deployed. This concern motivates the following section, which examines speculative computing, a paradigm that proactively reuses previously computed results to reduce redundant processing, as a complementary strategy for reducing computational load in resource-constrained CPS deployments.

2.5 Speculative Computing in Resource-Constrained Systems

Edge devices deployed in CPSs, such as single-board computers, low-power micro-controllers and embedded Internet of Things (IoT) nodes, operate under significant and often non-negotiable resource constraints [34], [35], [36]. Unlike server-grade or cloud infrastructure, these platforms

are limited to single or dual-core processors operating at low clock frequencies, with restricted memory, limited storage and tight energy budgets, particularly in battery-powered or thermally constrained deployments [35], [37]. Resource management in fog and edge environments is therefore a first-class engineering concern, as the heterogeneous and dynamic nature of edge hardware makes it impossible to rely on the abundant compute and memory availability assumed by cloud-oriented software architectures [34], [36], [37], [38].

These hardware limitations are compounded by the operational demands of CPS control software. Real-time control loops must complete within fixed time budgets to maintain synchronization with physical processes and deadline violations can result in degraded control performance or system instability [10], [37]. In distributed CPS architectures, a single edge device may host multiple concurrent control pipelines, each comprising sequences of function blocks performing sensor acquisition, signal processing, state estimation and actuation [10], [34], [36]. The cumulative computational demand of these pipelines can approach or exceed the processing capacity of resource-constrained edge hardware, particularly when individual pipeline stages involve computationally intensive operations such as statistical aggregation, energy estimation or filtering over sliding windows [34], [35]. The challenge is therefore not only to implement correct control logic, but to do so within the computational envelope of the target device without sacrificing responsiveness or accuracy [35], [37].

One computational paradigm that addresses this challenge directly is speculative computing, which proactively initiates computations based on predicted inputs in order to eliminate redundant processing on the critical execution path.

Speculative computing is a paradigm in which a system proactively initiates computations before all necessary inputs are confirmed, based on predictions about likely future states or values [39]. Rather than waiting for complete information before beginning computation, a speculative system makes a prediction, computes a result under that prediction and later validates whether the prediction was correct [39]. If validation succeeds, the precomputed result is committed and latency is effectively eliminated from the critical path; if validation fails, the speculative result is discarded and computation proceeds conventionally, incurring only the bounded overhead of the failed speculation [35], [39].

The paradigm originates from processor micro architecture, where branch prediction and out-of-order execution allow processors to speculatively execute instructions along a predicted control flow path before branching conditions are resolved [39], [40]. Thread-Level Speculation (TLS) generalizes this concept to software: sequential code fragments are executed optimistically in parallel by different threads, with dependence violations detected and corrected at runtime [39]. Beyond micro architecture, the speculative paradigm has been extended to higher levels of the computing stack, operating systems, compilers, databases and distributed systems, wherever the cost of misprediction is bounded and the expected gain from correct speculation is significant [35], [39].

In the specific context of resource-constrained edge systems, memoization offers a particularly tractable realization of this paradigm: rather than speculating about future control flow, the system

caches and reuses the results of expensive computations when inputs repeat a defined tolerance.

In resource-constrained edge systems, a particularly effective form of speculative reuse is memoization: the result of a computationally expensive function is cached alongside the input values that produced it and reused in future invocations when inputs match within a defined criterion [35]. A key insight enabling this approach is that many deployed applications, from system software to control pipelines, exhibit structural stability: despite dynamic runtime inputs, their computational patterns are sufficiently regular and predictable that stable sub-computations can be identified and selectively reused [35]. On the edge, where both compute and memory are scarce, the viability of memorization depends on the ability to identify these stable elements and cache them judiciously, avoiding indiscriminate reuse that would exhaust available memory [35], [37].

In sensor-driven CPS pipelines, physical processes such as temperature, humidity or energy consumption exhibit temporal continuity: they evolve slowly relative to sensor sampling rates, producing successive control cycles with minimally differing inputs [10], [34]. For deterministic computations, where identical inputs always produce identical outputs, this implies that cached results may remain valid across many consecutive cycles [35]. By checking whether current inputs match a cached entry within a tolerance reflecting sensor resolution or control precision, a system can return the cached result immediately on a hit and update the cache only on a miss [35]. This trades a small and bounded memory overhead for potentially substantial reductions in Central Processing Unit (CPU) utilization across the majority of cycles [35], [37].

Reducing computational overhead in resource-constrained edge systems has been approached from several directions. Resource management surveys identify computation offloading, scheduling and caching as primary strategies for mapping the constrained nature of fog and edge hardware [34], [37]. At the architectural level, speculative parallelism techniques such as TLS exploit predicted independence between computation fragments to reduce execution latency [39].

These approaches address computation optimization as a concern of the underlying system infrastructure, separate from and transparent to the control application logic. The integration of memoization-based speculative reuse directly within IEC 61499 control application abstractions, exploiting domain-specific knowledge about the temporal continuity of physical sensor signals, has not been investigated.

The practical benefit of this approach depends, however, on the statistical properties of the signals and computations involved, which determine how often cached results can be validly reused.

The effectiveness of memoization-based speculation depends on the statistical properties of the input signal and the tolerance used for cache key matching [35]. For slowly varying physical signals a tolerance, calibrated to the precision of the sensor or the resolution requirements of the control application can yield high hit rates with negligible loss of output fidelity [34], [35]. However, for highly variable, stochastic or event-driven inputs, successive values may differ substantially, resulting in frequent cache misses and low hit rates. In such cases, the overhead of cache management and lookup may outweigh the computational savings [35], [39].

Memoization-based speculative reuse is therefore most beneficial for pipelines with deterministic computations, slowly varying inputs and periodic execution patterns, characteristics commonly found in environmental monitoring, HVAC control, energy management and similar applications within CPSs [34], [35]. In these contexts, it represents a lightweight and deployment-transparent optimization that can substantially reduce the CPU demand of computationally intensive control pipelines on resource-constrained edge hardware, without requiring modifications to the underlying control logic or RTE [35], [37].

2.6 Research Gap

The literature reviewed across this Chapter covers three areas: the IEC 61499 execution model and its deployment through RTEs, the role of CFBs in hierarchical system modeling and the application of speculative computing to resource-constrained edge deployments. Table 2.1 summarizes the key contributions of existing work and their relationship to the research objectives of this thesis.

Table 2.1: Summary of related work and research gaps addressed by this work. Columns indicate whether each contribution addresses CFB execution overhead characterization (CFB overhead), memoization-based speculative reuse within IEC 61499 (Memo.), and empirical deployment design Guidelines. ✓ = addressed; × = not addressed.

Work	**Area**	**CFB overhead**	**Memo. in IEC 61499**	**Guidelines**
Lyu *et al.* [4]	IEC 61499 automation	×	×	×
Lyu & Brennan [6]	Multi-agent CPS	×	×	×
Sonnleithner *et al.* [10]	CFB design patterns	×	×	✓
Hopsu *et al.* [9]	IEC 61499 portability	×	×	×
Xavier *et al.* [16]	Formal verification	×	×	×
Gao *et al.* [12]	Execution semantics	×	×	×
Dai *et al.* [13]	Concurrent semantics	×	×	×
Wiesmayr *et al.* [11]	Model-based execution	×	×	×
Ramanujam [35]	Edge memoization	×	×	×
Hong & Varghese [37]	Edge resource mgmt.	×	×	×
This thesis	CFB overhead + memoization	✓	✓	✓

Three gaps emerge from this analysis. First, while CFBs are extensively studied as structural abstractions for hierarchical modeling and encapsulation [6], [10], the computational overhead introduced by hierarchical composition in practice, in terms of CPU usage, memory footprint and deployment latency, has not been systematically characterized. Second, memoization-based speculative reuse has been demonstrated as an effective technique for reducing computational load in general edge systems [35], but its application within IEC 61499 control application abstractions has not been investigated. Third, no empirical design guidelines exist for the effective use of CFBs in IEC 61499 deployments on resource-constrained edge hardware.

The combination of these three gaps defines the scope of this thesis. By characterizing the trade-offs of CFB-based hierarchical composition in a Python-based RTE, investigating memoization as a mitigation strategy at the CFB level and deriving practical design guidelines from experimental results, this work contributes knowledge that is absent from the current literature.

The literature reviewed across this chapter establishes the theoretical and contextual foundations for the work presented in this thesis. The gaps identified, the absence of empirical characterization of CFB overhead, the unexplored application of memoization within IEC 61499 abstractions and the lack of practical design guidelines, define the scope of the investigation. Chapter 3 presents the methodology designed to address these gaps.

Chapter 3

Methodology

This chapter presents the proposed solution for investigating the use of IEC 61499 CFBs as a structuring and optimization mechanism in distributed CPS pipelines deployed on resource constrained edge-hardware. Building on the theoretical foundations established in Chapter 2, the solution is designed to address three complementary objectives:

1. characterizing the trade-offs introduced by hierarchical CFB composition in terms of computational overhead and observability;
2. evaluating whether memoization-based speculative reuse can mitigate the performance disadvantages associated with CFB encapsulation;
3. deriving practical guidelines for the effective use of CFBs in edge-deployed IEC 61499 applications.

To achieve these objectives, a distributed pipeline architecture is proposed comprising two resource-constrained edge nodes and a coordination host, interconnected through standard industrial communication protocols. The architecture supports two configurations under evaluation, a baseline configuration using a flat FB network and a CFB-based configuration incorporating a memoization-based caching mechanism, enabling direct empirical comparison of their respective computational profiles.

The experimental evaluation follows a two-phase approach. In the first phase, a focused pipeline with a small number of FBs is used to isolate and characterize the fundamental trade-offs, introduced by CFB encapsulation and memoization-based caching under controlled conditions. In the second phase, the same evaluation is repeated with a more complex pipeline comprising a larger number of FBs, with the dual purpose of validating the conclusions drawn from the first phase and examining how the observed trade-offs scale with the pipeline complexity, including metrics such as deploy time, CPU utilization, memory consumption as the number of encapsulated FBs increases.

A dedicated monitoring subsystem captures CPU utilization and memory consumption through each experiment, providing the quantitative basis for the evaluation presented in Chapter 5.

The remainder of this chapter is organized as follows. Section 3.1 introduces the target RTE and IDE and the rational for their selection. Section 3.2 describes the distributed pipeline architecture. Section 3.3 defines the pipeline configurations under evaluation. Section 3.4 presents the memoization-based caching mechanism and its integration within CFB structure. Section 3.5 describes the system monitoring and metric collection infrastructure.

3.1 Target RTE and IDE

This sections present the two tools that form the implementation platform of this thesis the DINASORE RTE and the 4diac IDE.

3.1.1 DINASORE RTE

The RTE selected for this work is DINASORE, a Python-based framework that implements the IEC 61499 standard for FB-based control of CPPS [29], [41]. DINASORE provides data integration with third-party platforms through OPC UA, enabling runtime monitoring and interaction with deployed FB instances and adopts the 4diac IDE as its graphical interface for the design and deployment of FB networks [29], [41].

DINASORE was selected as the target RTE for several reasons. Its Python-based implementation allows the RTE itself to be extended and modified, which is necessary for integrating CFB support and memoization-based caching mechanisms directly into the execution engine. The use of Python also enables seamless integration of data processing and communication libraries within FB algorithms, supporting the computational intensive pipeline stages used in the evaluation. Furthermore, DINASORE is explicitly designed for deployment on resource-constrained edge hardware in CPS environments, making it a suitable platform for investigating the overhead characteristics of CFB-based applications [41].

It is acknowledge that DINASORE is a research-oriented RTE and does not provide the hard real-time scheduling guarantees of production RTEs such as FORTE [9], [27]. However, as the objectives of this work concern the characterization of computational trade-offs rather than the validation of real-time control performance, DINASORE provides an appropriate platform for the experimental investigation carried out in this thesis.

It is important to note that the original DINASORE framework does not natively support the execution of CFBs. In its original form, DINASORE is capable of deploying and executing networks of FBs, but lacks the execution engine require to instantiate and resolve the internal FB networks of CFBs at runtime. Extending DINASORE with CFB execution support is therefore a prerequisite of the proposed solution and constitutes one of the contributions of this work. The design of this extension is described in Chapter 4, where the implementation of the CFB execution engine and its integration into the existing DINASORE architectures are presented in detail.

3.1.2 4diac IDE

The Eclipse-based 4diac IDE is used as the graphical engineering environment for the design and deployment of IEC 61499 applications in this work [7], [32]. The 4diac IDE provides comprehensive support for graphical modeling of FBs and CFBs, definition of event and data connections, library management and device and resource configuration [7], [9], [32]. Its system configuration capabilities allow FB instances to be mapped to specific devices and resources across distributed architecture and deployment configurations to be generated and sent directly to target RTEs [9].

The 4diac IDE was selected due to its native integration with DINASORE, its mature support for CFB modeling and hierarchical system decomposition, configuring and deploying distinct pipeline configurations across multiple edge nodes from a consistent engineering environment.

3.2 Distributed pipeline architecture

The proposed solution adopts a distributed pipeline architecture in which the processing workload is partitioned across multiple heterogeneous nodes, each running an independent instance of the DINASORE RTE. This architecture reflects the operational characteristics or real-world CPS deployments, where sensor data is generated at geographically distributed locations and must be processed locally [34] before results are transmitted upstream [34], [37].

The physical infrastructure consists of two Raspberry Pi edge nodes and a management node running on a standard PC. Each edge node is responsible for acquiring sensor data from its local environment, executing the computationally intensive processing stage and publishing the resulting output to a message broker. The management node runs a lightweight receiving pipeline that subscribes to the data published by the edge nodes, acting as the downstream consumer of the distributed data flow. In addition, the management node hosts the 4diac IDE, which is used to deploy and configure the FB network across all nodes and additionally runs the monitoring infrastructure used to collect performance metrics during the evaluation phase.

Figure 3.1 illustrates this two-layer structure showing how each physical node maps to a dedicated CFB that encapsulates its local processing pipeline.

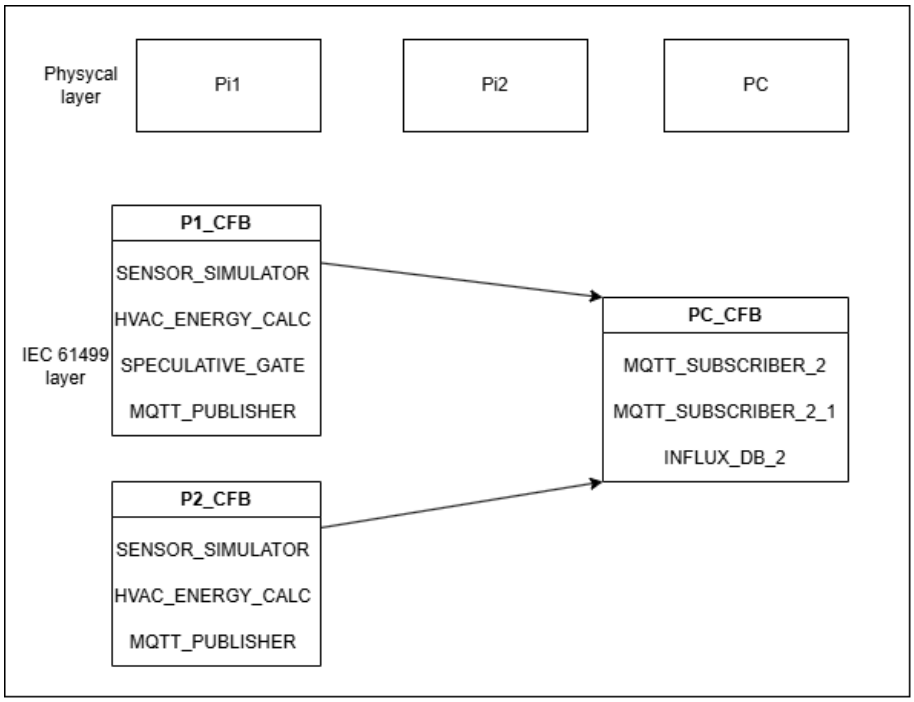


Figure 3.1: Diagram of the CPPS.

The processing pipeline on each node follows a linear data flow structure consisting of three functional stages: sensor data acquisition, signal processing and data dissemination. A sensor simulation FB produces synthetic readings at a fixed interval; a computational FB applies an energy estimation algorithm to the incoming data, representing a resource-intensive workload typical of edge CPS scenarios; and a publisher FB transmits the processed results via Message Queuing Telemetry Transport (MQTT) [42]. This three-stage structure is intentionally representative of a broad class of industrial CPS pipelines, ensuring that the evaluation results are generalizable beyond the specific application domain used. Figure 3.2 illustrates the communication architecture connecting these pipeline stages across the distributed nodes.

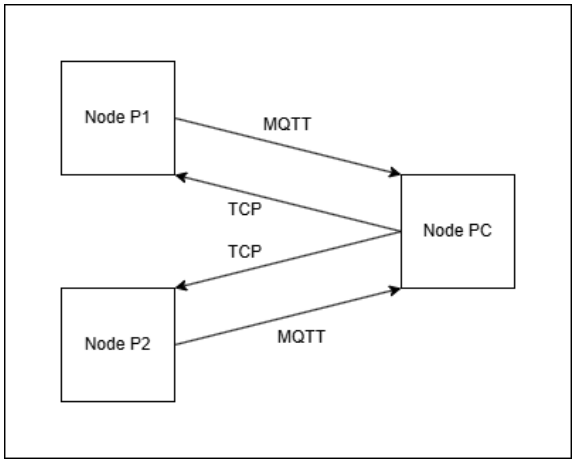


Figure 3.2: Diagram of the distributed architecture.

Inter-node communication and FB network management rely on two complementary protocols. The 4diac IDE communicates with each DINASORE instance via Transmission Control Protocol (TCP), using the standard IEC 61499 management protocol to deploy FB instances, establish event and data connections and initiate execution [41]. In parallel, DINASORE exposes the internal state of active FBs through an OPC UA server [31], enabling external clients to monitor variable values at runtime. This separation between a management plane TCP and an observability plane (OPC UA) is consistent with established practices in distributed IEC 61499 deployments [9], [10].

3.3 Pipeline Configuration

To evaluate the impact of the proposed CFB-based memoization-based caching mechanism, two pipeline configurations are defined. Both configurations implement the same logical data flow, sensor data acquisition, computational processing and data dissemination, and are deployed on the same physical infrastructure, ensuring that any measured differences in performance are attributed to the structural and execution differences between configurations rather than hardware or workload variation.

The first configuration, referred to as Baseline, represents the reference implementation. It consists exclusively of FBs connected in a linear sequence: a sensor simulator block produces synthetic readings at a fixed interval, a computation block applies a resource-intensive processing algorithm to the incoming data and a publisher block transmits the result to an external message broker. This configuration server as the control condition for the evaluation, establishing the performance and resource consumption characteristics of a conventional IEC 61499 pipeline without any memoization-based optimization.

The CFB boundary further enables a second benefit beyond encapsulation: it provides a natural interception point at which a caching mechanism can be introduced without modifying the constituent FBs. The *SPECULATIVE_GATE* exploits this property, sitting at the entry of the computationally intensive stage and returning cached results transparently when input conditions permit.

The second configuration, referred to as full-cfb, is the proposed configuration and constitutes the core contribution of this work. It encapsulates the same three functional stages within a CFB and introduces memoization-based caching mechanism, the *SPECULATIVE_GATE*, at the entry point of the computationally intensive processing stage. Once a given input value and its corresponding output have been observed together a configurable number of times, subsequent occurrences of the same input cause the cached result to be returned directly, bypassing the computation stage entirely. When no suitable cache entry exists, the block executes normally and stores the result for potential future reuse. The encapsulation of this logic within a CFB allows the mechanism to be deployed as a self-contained, reusable unit without modifying the internal logic of the constituent FB. Figure 3.3 illustrates the two pipeline configurations side by side.

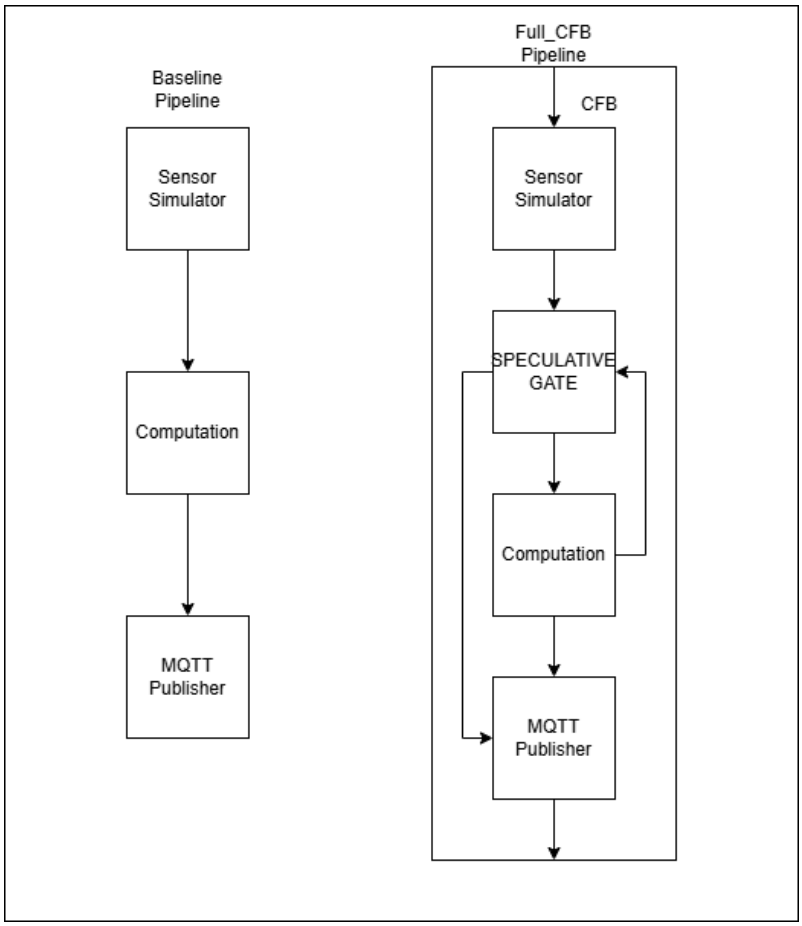


Figure 3.3: Diagram exemplifying both configurations.

Both configurations apply consistently across all nodes. In the baseline, all nodes, the two edge nodes and the management node, run their respective pipelines using FBs. In the full-cfb configuration, each node encapsulates its pipeline within a dedicated CFB: the edge nodes introduce the *SPECULATIVE_GATE* mechanism within their CFB, while the management node encapsulates its receiving pipeline in a separate CFB without a caching component, since it performs no computation-intensive processing.

The two configurations are evaluated under identical workload conditions on the same physical nodes, enabling a direct comparison of CPU utilization, memory consumption and deploy time. The full_cfb configuration is additionally evaluated with respect to the trade-offs introduced by CFB encapsulation, particularly the reduction in OPC UA of observability of internal FB state.

3.4 Memoization-based Caching Mechanism

The memoization-based caching mechanism introduced in the full-cfb configuration is implemented through a dedicated functional component, the *SPECULATIVE_GATE*, positioned at the

entry point of the computationally intensive processing stage within the CFB. Its role is to intercept incoming sensor values, determine wether a reliable cached result exists for that input and either return the cached output or forward the input to the computation block for full evaluation.

The mechanism operates on the principle that the confidence in a cached result must be established before it can be safely reused [35], [43]. Rather than caching a result on the first observation, the *SPECULATIVE_GATE* tracks how many times each input value and its corresponding output have been observed consecutively. Only once this count reaches a configurable threshold does the mechanism activate memoization for that input: from that point forward, subsequent occurrences of the same value cause the cached output to be returned directly, bypassing the computation stage entirely. If the input changes, the count resets and the evaluation process begins again. Figure 3.4 shows the decision flow of the *SPECULATIVE_GATE* mechanism

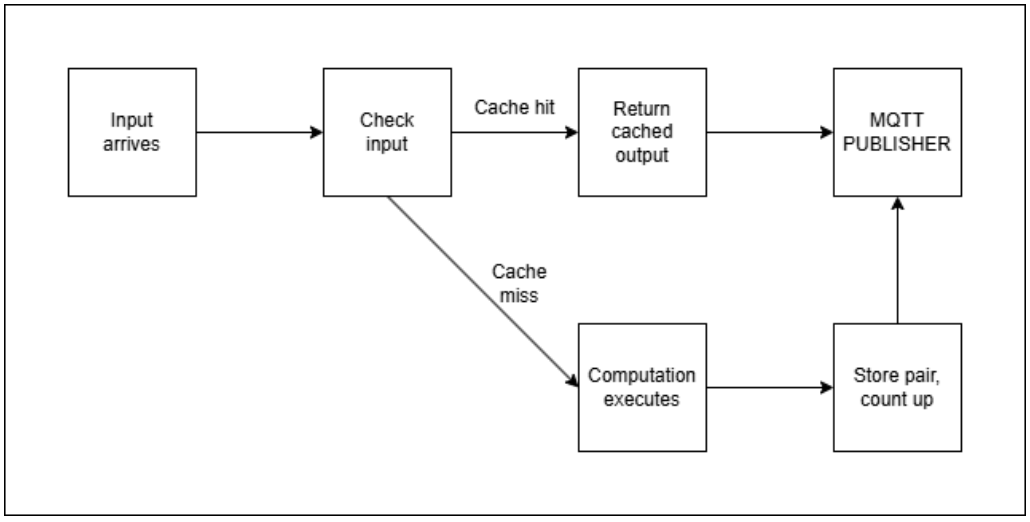


Figure 3.4: Flowchart of the SPECULATIVE-GATE.

This design is motivated by the structural stability property of sensor-driven CPS pipeline, discussed in Section 2.5. In environments where physical quantities vary slowly relative to the sampling rate, consecutive readings frequently share the same or similar values, making repeated full computation redundant [35]. The confirmation threshold introduces a controlled delay before memoization is activated [35], reducing the risk of caching transient or anomalous readings and ensuring that only stable, recurring input patterns benefit from the result reuse.

The *SPECULATIVE_GATE* is deliberately designed as a self-contained component with a well-defined interface, agnostic to the specific computation it precedes. This allows it to be reused across different CFB configurations without modification and positions it as a general-purpose building block for memoization-based optimization in IEC 61499 pipelines [43].

3.5 System Monitoring and Metrics Collection

To support a rigorous comparative evaluation of the pipeline configuration, the proposed solution incorporates a dedicated monitoring subsystem that operates alongside the DINASORE RTE on each node. This subsystem collects quantitative performance metrics throughout the execution of each configuration, storing them in a time-series database for subsequent analysis and visualization.

Four metrics are defined for the evaluation. CPU utilization and memory consumption are measured continuously on each node, capturing the resource overhead imposed by each configuration under steady-state operation. These metrics are particularly relevant in the context of edge CPS deployments, where hardware resources are constrained and sustained high utilization directly affects system reliability and energy consumption [34], [37]. Deployment time, the interval between the instantiation of the first and last FB in the network, is recorded for each configuration, providing a measure of the additional overhead introduced by CFB encapsulation during the initialization phase. Finally, the in OPC UA observability resulting from CFB encapsulation is documented as a qualitative trade-off: whereas all FB in the baseline configuration expose their internal state individually as OPC UA nodes, the full-cfb configuration, in theory, exposes only the external interface of each CFB, making the internal FB network invisible to external monitoring clients.

The monitor subsystem is designed to be non-intrusive with respect to the pipeline under evaluation. It runs as a background process on each node, periodically sampling system-level metrics independently of the FB execution cycle, ensuring that the act of measurement does not itself introduce measurable overhead into the pipeline. Collected data is written to a centralized time-series database and visualized through a real-time dashboard, enabling cross-node comparison and later analysis.

This Chapter presented the methodology adopted to investigate the research question, covering the target RTE and IDE, the distributed pipeline architecture, the two configurations under evaluation and the monitoring infrastructure. The design decisions described here, from the choice of DINASORE and 4diac to the structure of the *SPECULATIVE_GATE* and the metrics collected, provide the basis for the implementation described in Chapter 4.

Chapter 4

Implementation

4.1 Hardware and Software Setup

This section describes the hardware and software infrastructure use in the implementation, covering the physical testbed, the software dependencies and the modifications to the DINASORE.

4.1.1 Hardware Infrastructure

The experimental infrastructure consists of three networked nodes. Two Raspberry Pi 3 Model B single-board computers serve as edge processing nodes, designated Node P1 and Node P2, connected to the local network. Each Raspberry Pi 3 Model B [44], [45] is equipped with a quad-core ARM Cortex-A53 processor clocked at 1.2Ghz and 1Gb of Random Access Memory (RAM), representative of the resource-constrained hardware typical of edge CPS deployments. The management and monitoring node is an ASUS TUF GAMING A15 laptop running Windows11, connected to the same WiFi wireless network as the edge nodes. Remote access to the Raspberry Pi nodes was performed from the management node via RealVNC, enabling terminal interaction for starting the DINASORE RTE and rebooting the nodes. File transfer between the management node and the edge nodes were performed via Security Copy Protocol (SCP) from the PC terminal. Figure 4.1 provides a complete view of the distributed architecture, including all three nodes and their communication links.

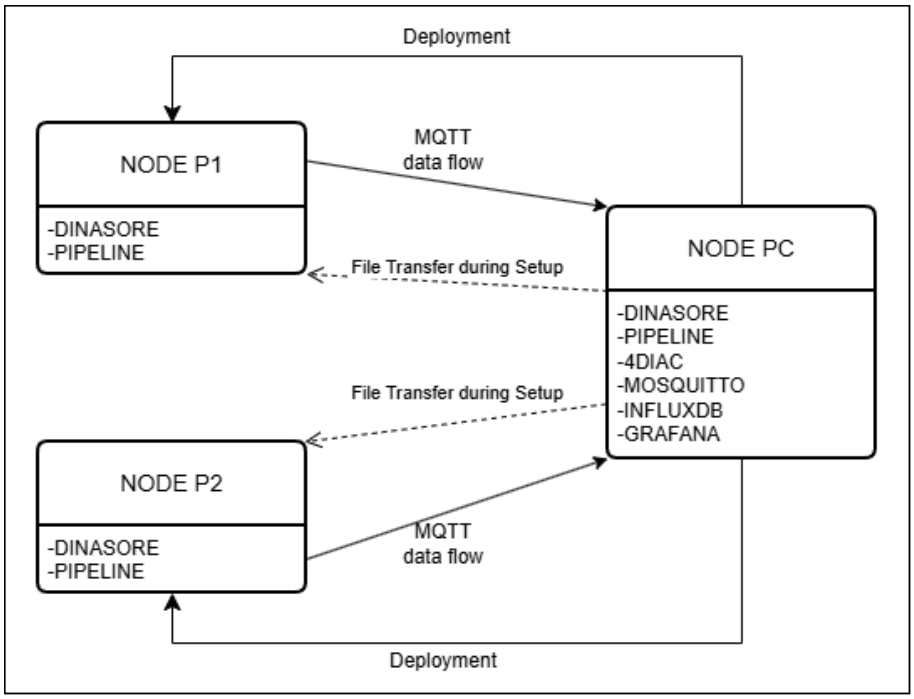


Figure 4.1: Complete diagram of the distributed architecture.

4.1.2 Software Stack

Each Raspberry Pi node runs a modified instance of DINASORE under Python 3, with two additional libraries installed: `paho_mqtt` [46], used by the MQTT publisher FB for message transmission and influxdb [47], used by the monitoring subsystem to write performance metrics to the time-series database. Mosquitto was selected as the MQTT broker for its lightweight footprint and wide adoption in IoT deployments [42], [48]. InfluxDB [47] was chosen for its native support for time-series data and its Python client library and Grafana [49] for its direct integration with InfluxDB and support for real-time refresh.

The management node runs the following services:

1. The 4diac IDE for FB network design and deployment;
2. A Mosquitto MQTT broker that receives the data streams published by the edge nodes;
3. An InfluxDB instance accessible at port 8086 under the database name sensor-data, used to store both pipeline data and performance metrics;
4. A Grafana instance for real-time visualization of collected metrics;
5. And its own DINASORE instance to host the receiving pipeline.

4.1.3 DINASORE Modifications

The original DINASORE framework supports only the execution of FB: all FB logic was handled by a single base class `fb.py`, with no concept of a CFB whose behavior is itself defined

by a internal network of FB. Extending DINASORE to support FB execution required changes across six modules, spanning the execution model, the deployment and configuration layer and the OPC UA [31] integration.

Shared Base Class

The *FBInterface* base class was extended with two new attributes: `parent_config`, which holds a reference to the `Configuration` object that owns the FB instance and `is_composite`, a boolean flag defaulting to False and overridden by the composite subclass. A new method `get_full_path()` was added, which constructs the hierarchical path of a FB instance by traversing the chain of `parent_config` references, for example, returning `EMB_RES.SENSOR_PIPELINE.SPECULATIVE_GATE` for a FB nested inside a CFB. This path is used throughout the system whenever a FB's identity must be expressed relative to the full deployment hierarchy. Listing 4.1 shows the implementation of `get_full_path()`.

Listing 4.1: Hierarchical path resolution in `FBInterface`.

```
def get_full_path(self):
    if self.parent_config is None:
        return self.fb_name
    parent_path = (
        self.parent_config.get_full_path()
        if hasattr(self.parent_config, 'get_full_path')
        else self.parent_config.config_id
    )
    return f"{parent_path}.{self.fb_name}"
```

The implementation relies on a recursive traversal: if the current FB has a `parent_config`,it first resolves the parent's full path and appends the FB's own name; otherwise it returns the FB name alone, anchoring the recursion. This design ensures that path resolution remains correct regardless of nesting depth and requires no changes to the constituent FBs or their algorithms. The `is_composite` flag and `parent_config` reference, together with `get_full_path()`, form the minimal extension to `FBInterface` required to support hierarchical deployment. Figure 4.2 shows the resulting class hierarchy, with `FBInterface` as the shared base class.

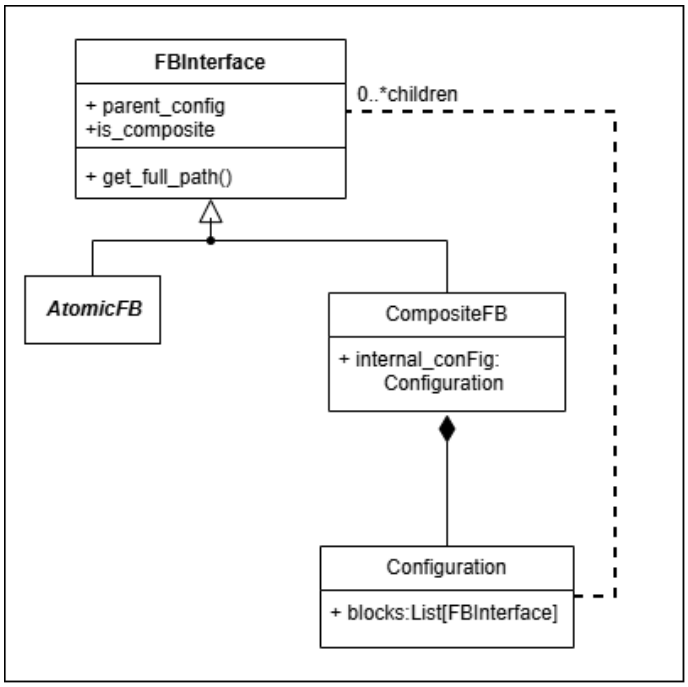


Figure 4.2: UML class diagram of *FBInterface* hierarchy.

Execution model

The original FB class was refactored into two specialized subclasses. `AtomicFB` (`core/fb_atomic.py`) preserves the original execution behavior: it waits for an incoming event, reads its input variables, executes the `schedule()` method defined in the FB´s Python module and updates its outputs. It additionally accepts a `parent_config` reference, enabling hierarchical path resolution.

CFB `core/fb_composite.py` is the central new component. On initialization, it parses the `FBNetwork` section of the CFB's .fbt declaration and instantiates the constituent FB inside a private `Configuration` object `self.internal_config`, supporting arbitrarily nested CFBs recursively. The internal connections are classified into three types:

1. Internal-to-internal connections (linked directly between sub-FB);
2. External-to-internal input mappings;
3. Internal-to-external output mappings;

At runtime, each incoming event on the CFB´s external interface triggers the application of input mappings, execution of the relevant sub-FB and propagation of outputs via output mappings. The coordination logic includes detection of the "winning" sub-FB when multiple internal paths can produce the same external output event, the pattern used by the *SPECULATIVE_GATE*, where both the cache hit and cache miss paths can produce a DONE event, but never simultaneously. Figure 4.3 shows the execution flow of the original DINASORE runtime engine, prior to the modifications described in this section.

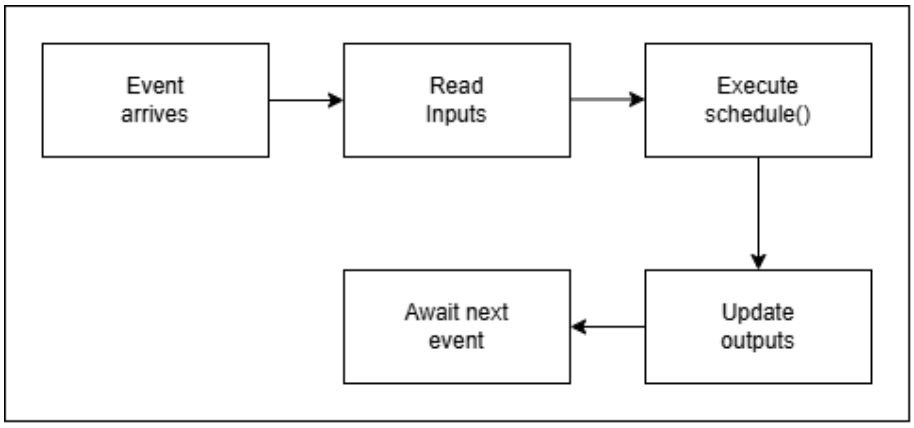


Figure 4.3: Execution flow of the original DINASORE runtime engine.

Figure 4.4 illustrates the updated flow, showing the new branching logic between *AtomicFB* and CFB execution paths.

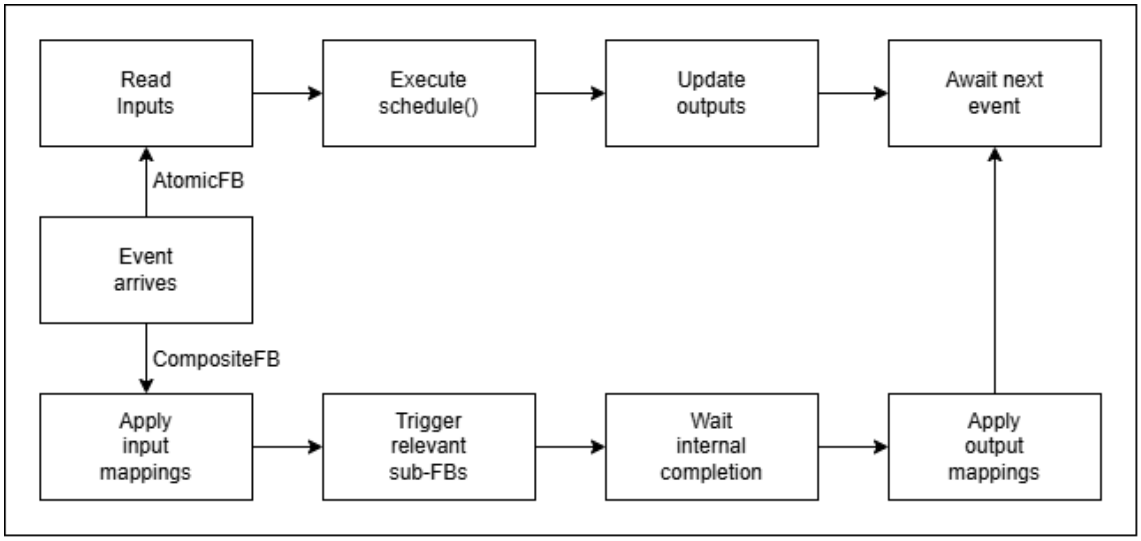


Figure 4.4: Execution flow of the extended DINASORE runtime engine, with separate branches for *AtomicFB* and *CompositeFB* execution.

CFB Detection

The `FBResources` module was extended with a new method, `is_composite_fb()`, which inspects the .fbt Extensive Markup Language (XML) to determine wether a FB type contains an `FBNetwork` element. The `import_fb()` method was updated to return early for CFB types without importing a Python module, since a CFB's behavior is defined entirely by its internal FB network, no Python implementation file exists or is needed. The `exists_fb()` method was updated accordingly: for CFB, only the .fbt file is required, for `AtomicFB`, both .fbt and .py must be present. Listing 4.2 shows the implementation of `is_composite_fb()`.

Listing 4.2: CFB detection in FBResources.

```
def is_composite_fb(self):
    xml_root = self.get_xml()
    if xml_root is None:
        return False
```

Configuration and Hierarchical Addressing

The `Configuration` class underwent the most extensive modifications. It now accepts a parent_config parameter and exposes `get_full_path()` enabling any configuration node in the hierarchy to identify itself by its full path. Two new methods were added: `create_composite_fb()`, which instantiates a CFB and registers it in the configuration and _resolve_connection_path(), which resolves dotted path strings such as `CFB.FB.ATTR` by navigating through nested `internal_config` objects to locate the target FB and attribute. The `locate_target()` method provides the same traversal for arbitrary full paths.

Two additional correctness fixes were introduced. First, `create_virtualized_fb()` and `create_connection()` were made idempotent through duplication guards. DINASORE processes deployment via two independent paths (live TCP commands from 4diac and a replay of the `data_model.fboot` file for OPC UA initialization), which previously caused the same FB instances and connections to be created twice, leaving orphaned threads with their own state but disconnected from the active network. Second, the `read_watches()`, `start_work()` and `stop_work()` methods were made recursive, descending into the `internal_config` of any CFB so that monitoring, startup and shutdown cover the full FB tree and not only the top-level configuration. Figure 4.5 shows the Idempotency guard added to `create_virtualized_fb()` to prevent duplicate FB creation.

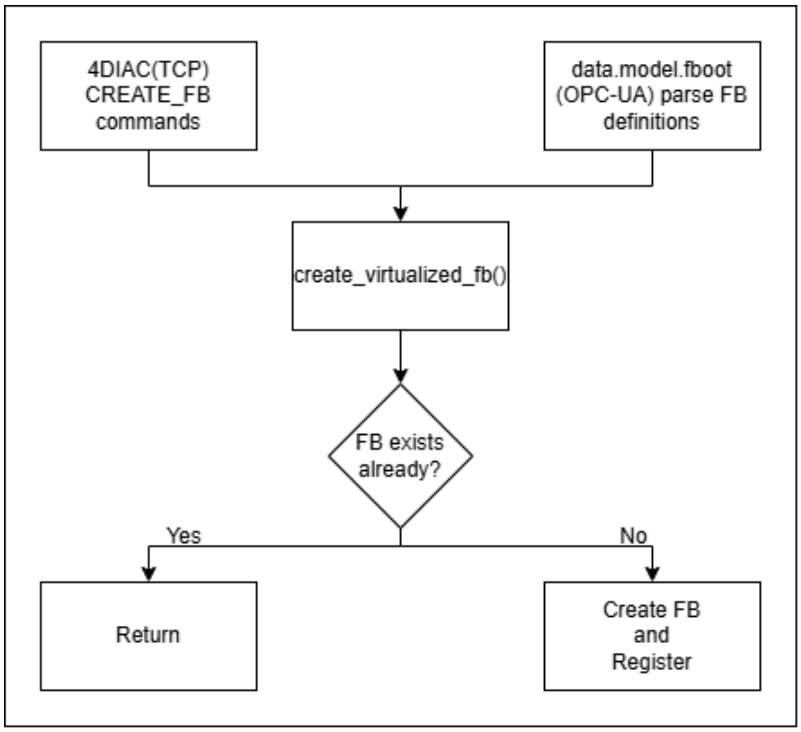


Figure 4.5: Idempotency guard in `create_virtualized_fb()`, preventing duplicate OPC UA node creation on repeated deployment requests.

A third correctness issue was identified in the output mapping propagation logic of `CompositeFB`: internal events were begin re-emitted to the external interface on every execution cycle, even when they had not changed since the previous cycle. This caused persistent event values, such as the *INIT_O* signal of *SENSOR_SIMULATOR*, which fires only once at initialization but remains readable indefinitely, to be incorrectly interpreted as new events in subsequent cycles. The fix captures a baseline snapshot of all relevant internal event values before dispatching each cycle and only propagates an event externally if its value has changed relative to that baseline.

Deployment Routing

The Manager class was extended with `self.hierarchy_index`, a dictionary mapping hierarchical path strings to their corresponding `Configuration` objects and `_locate_configuration_by_path()`, which descends through nested CFB to find the correct configuration for any incoming deployment command. The `parse_configuration()` method now uses this resolution to correctly route FB creation and connection commands targeting FBs nested inside CFBs, delegating to `create_composite_fb()` or `create_fb()` based on the result of `is_composite_fb()`. Figure 4.6 shows the updated deployment routing logic introduced in the `Manager` class.

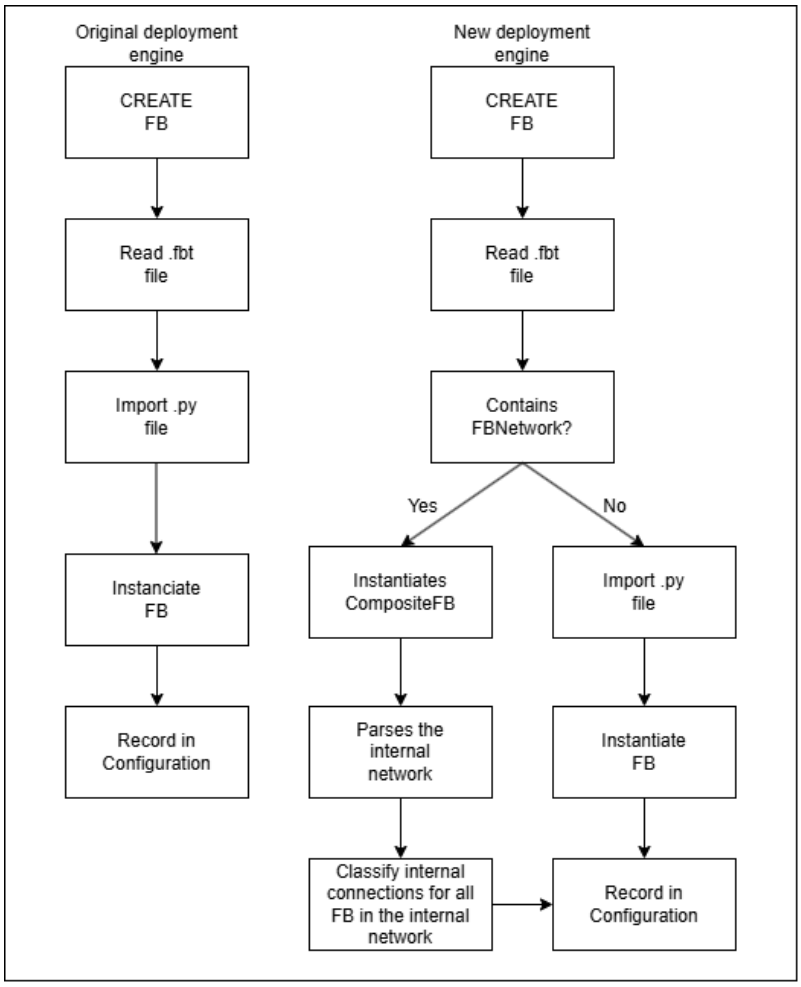


Figure 4.6: Deployment routing flow in the extended `Manager` class, dispatching CREATE requests to `AtomicFB` or `CompositeFB` handlers.

OPC UA Integration

The `generate_function_blocks()` method of `UaManagerFboot` was rewritten to extract hierarchical path prefix from each line of the .fboot file and locate the appropriate `Configuration` before creating the corresponding OPC UA [31] node. Without this change, FB nested inside a CFB would appear in the OPC UA node tree. A new helper `_locate_config_by_path()` provides the traversal and `_create_fb_ua_mode()` handles the distinction between atomic and composite types: atomic FB are parsed into OPC UA nodes via `parse_fbt()`, while CFB are logged only at the top level since their internal FB are addressed individually via their own hierarchical paths.

Speculative Computing: Design Decision

During development, two implementations of the memoization-based caching mechanism were explored. The first, `InputOutpuCache`, was implemented directly inside `CompositeFB`

as an engin-level cache configurable through attributes in the CFB's .fbt declaration, making it transparent to the pipeline designer. The second implemented the same logic as a dedicated FB, *SPECULATIVE_GATE*, placed explicitly within the CFB's internal network. The dedicated FB approach was adopted for the final pipeline: it keeps the caching logic visible as an explicit IEC 61499 component rather than hidden configuration of the runtime engine, making it easier to audit, test and explain in isolation. The engine-level implementation remains in the code base but is not used by the evaluation pipeline.

4.2 Atomic FB Design

This section describes each FB designed for the evaluation pipeline. Each FB implements a specific stage of the sensor-to-cloud processing workflow.

4.2.1 SENSOR_SIMULATOR

The *SENSOR_SIMULATOR* FB produces synthetic temperature readings by pre-computing a distribution of values at initialization and cycling through them sequentially at runtime. On receiving an *INIT* event, the FB generates a set of values following a normal distribution ($\mu = 0.5$, $\sigma = 0.1$) shifted by a configurable *OFFSET* parameter, set to 20, which defines the base temperature value around which readings vary. On each READ event, the next value in the distribution is returned and the block pauses for 1 second before re-triggering itself via the self-loop mechanism. The 1 second interval was chosen to produce a read rate representative of typical sensor polling in HVAC monitoring scenarios while remaining slow enough to favor structural stability and thus cache hit accumulation in the *SPECULATIVE_GATE*. The slow-varying nature of the generated readings, following the shape of a bell curve, reflects the structural stability property discussed in Section 2.5 and is what makes the memoization-based caching, mechanism possible and effective in this pipeline. Figure 4.7 shows the interface of the *SENSOR_SIMULATOR* FB.

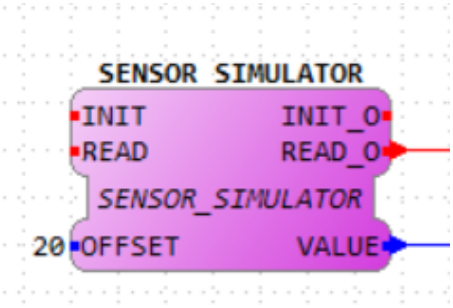


Figure 4.7: *SENSOR_SIMULATOR* FB.

4.2.2 HVAC_ENERGY_CALC

The *HVAC_ENERGY_CALC* FB implements the computing-intensive processing stage. It models an energy estimation algorithm for HVAC systems, computing estimated energy consumption

based on the incoming temperature reading and the configured floor area of the zone. Node P1 is configured with a floor area of 150m and Node P2 with 300m.

Computational intensity is controlled by two parameters: `N_ITER`, set to 20 and n, set to `max(WINDOW * 5000, 100000)`, which determines the number of iterations per cycle. These values were chosen to produce a sustained and measurable computational load on the Raspberry Pis hardware, representative of realistic edge CPS processing workloads. Figure 4.8 shows the interface of the *HVAC_ENERGY_CALC* FB.

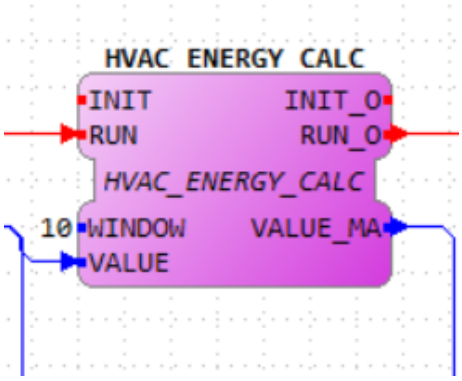


Figure 4.8: *HVAC_ENERGY_CALC* FB.

4.2.3 MQTT_PUBLISHER, MQTT_SUBSCRIBER and INFLUX_DB_2

The *MQTT_PUBLISHER_2* FB transmits the output of *HVAC_ENERGY_CALC* to the Mosquitto broker on the management node using `paho_mqtt` library. Each edge node publishes to a dedicated topic, allowing the management node to distinguish between the two data streams. The *MQTT_SUBSCRIBER_2* FB, deployed on the management node, subscribes to both topics and forwards received values into the local FB network. The *INFLUX_DB_2* FB writes the received values to the `sensor_data` InfluxDB database under the measurement Temp_read, tagged with the originating node identifier. Figures 4.9, 4.10 and 4.11 show the interfaces of the *MQTT_PUBLISHER_2*, *MQTT_SUBSCRIBER_2* and *INFLUX_DB_2* FBs, respectively.

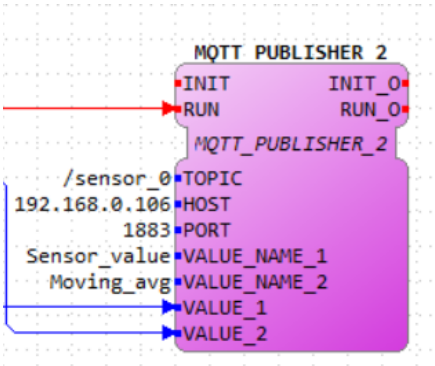


Figure 4.9: *MQTT_PUBLISHER_2* FB.

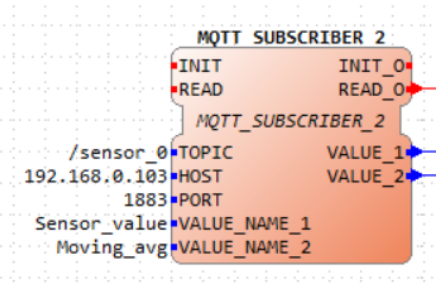


Figure 4.10: *MQTT_SUBSCRIBER_2* FB.

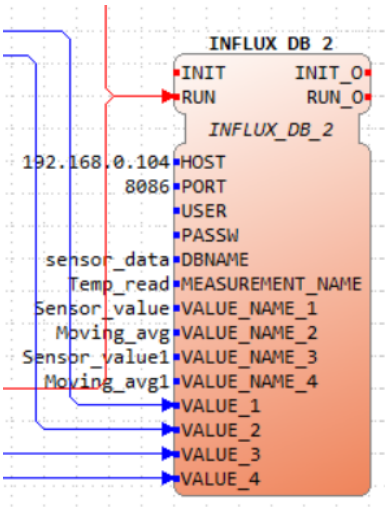


Figure 4.11: *INFLUX_DB_2 FB*.

4.3 Pipeline CFB

In the full_cfb configuration, each edge node's pipeline is encapsulated within a CFB containing four constituent FB: *SENSOR_SIMULATOR*, *SPECULATIVE_GATE*, HVAC_ENERGY_CALC and *MQTT_PUBLISHER_2*. The internal event routing is as follows: SENSOR_SIMULATOR emits to *SPECULATIVE_GATE*, which routes to *HVAC_ENERGY_CALC* on a cache miss or directly to *MQTT_PUBLISHER_2* on a cache hit. On a cache miss, *HVAC_ENERGY_CALC* returns the result to *SPECULATIVE_GATE*, which stores the pair and forwards to *MQTT_PUBLISHER_2*. Both paths produce a DONE event on the CFB's external interface, never simultaneously, a coordination pattern handled by the CFB execution engine as described in Section 4.1.3. Figure 4.12 shows the external interface of the *PIPELINE_P1* CFB as seen from 4diac.

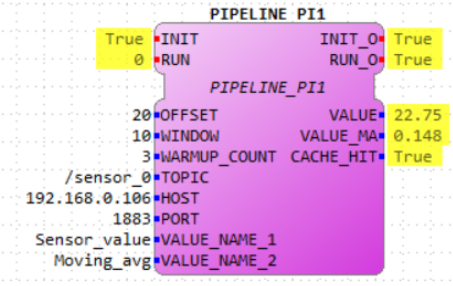


Figure 4.12: *PIPELINE_P1* CFB.

Figure 4.13 shows the internal FB network of *PIPELINE_P1* in the baseline configuration, without the *SPECULATIVE_GATE*.

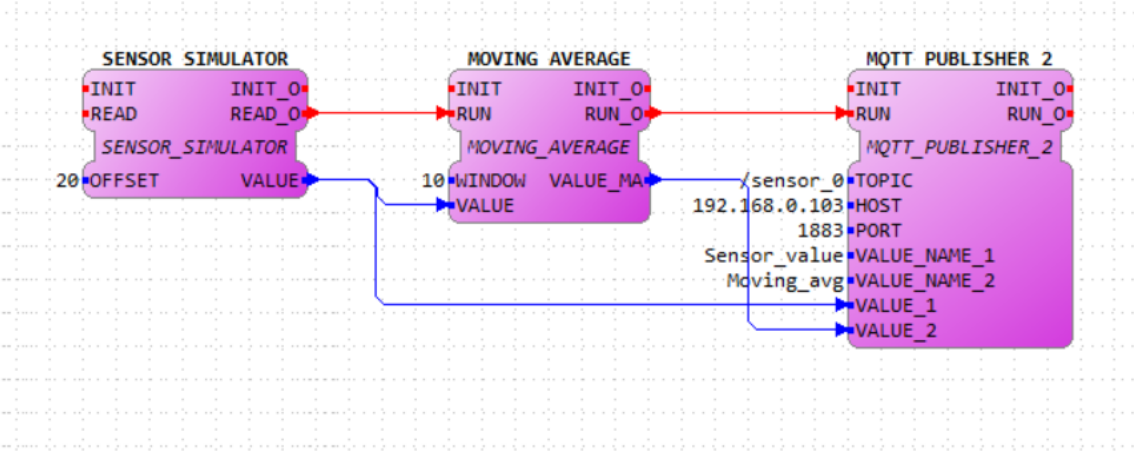


Figure 4.13: Internal FB network of *PIPELINE_P1* without *SPECULATIVE_GATE*.

On the management node, the receiving pipeline is encapsulated in *PIPELINE_PC*, a CFB containing two *MQTT_SUBSCRIBER_2* and *INFLUX_DB_2* without a caching component, as this node performs no computation-intensive processing. Figure 4.14 shows the external interface of the *PIPELINE_PC* CFB. The interface exposes two INIT and RUN event inputs, two *INIT_O* and *RUN_O* event outputs, and a set of data ports that aggregate the configuration parameters of the internal *MQTT_SUBSCRIBER_2* and *INFLUX_DB_2* FBs, including broker address, subscription topics, port, database credentials and value name mappings.

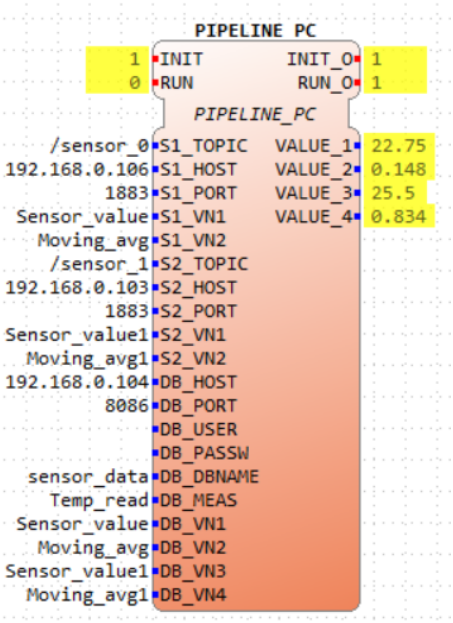


Figure 4.14: PIPELINE_PC CFB external interface, aggregating the configuration parameters of the internal MQTT_SUBSCRIBER_2 and INFLUX_DB_2 FBs into a single deployable unit for the management node.

Figure 4.15 shows the internal FB network of *PIPELINE_PC*. The two *MQTT_SUBSCRIBER_2*

instances subscribe to the dedicated topics published by `Pi1` and `Pi2` respectively, and forward received values to the *INFLUX_DB_2* FB, which writes them to the InfluxDB database under the configured measurement and node tags.

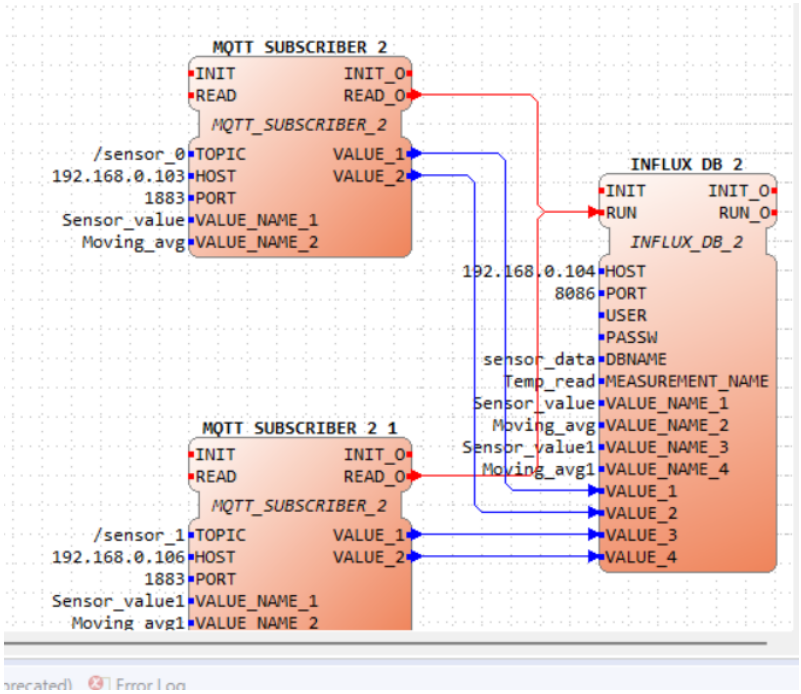


Figure 4.15: Internal FB network of PIPELINE_PC, showing two MQTT_SUBSCRIBER_2 instances subscribing to the edge node topics and forwarding data to a shared INFLUX_DB_2 FB for time-series storage.

4.4 Deployment with 4diac and DINASORE

The FB network for each configuration is defined in the 4diac IDE and exported as a .fboot file (`resources/data_model.fboot`), which encodes the complete network topology: FB instances, type assignments, parameter values and event and data connections. DINASORE parses this at startup and instantiates the FB network accordingly.

DINASORE is started locally on each node. On the edge nodes, a terminal session is opened via RealVNC and the following command is executed:

```
nohup python -B core/main.py -a <node_ip> -l INFO
--influx_host <PC_IP> > dinasore.log 2>&1 &
```

where <node_ip> is replaced by the static address of the respective node. The `nohup` and background operator ensure the process persists after RealVNC session is closed. On the management node, DINASORE is started from the PC terminal:

```
python -B core/main.py -l INFO --influx_host localhost
```

Once all instances are running, the 4diac IDE connects to each node via TCP and deploys the FB network by issuing a sequence of management commands: creating FB instances, setting parameter values, establishing event and data connections and issuing the initialization trigger. FB with no initialization trigger such as *SENSOR_SIMULATOR* are initialized automatically by the RTE during deployment.

Deployment time is derived from dinasore.log by extracting the timestamps of the first and last FB instantiation events, providing a measure of the additional initialization overhead introduced by CFB encapsulation relative to the baseline configuration.

4.5 Monitoring Infrastructure

This section describes the monitoring subsystem deployed alongside the DINASORE RTE to collect and visualize performance metrics during the experimental evaluation.

4.5.1 System Metrics Collection and Visualization

A dedicated module, `monitor_system.py`, runs as a daemon thread on each node alongside the DINASORE RTE. Every 5 seconds, it samples CPU utilization and memory consumption using `psutil` library and writes the values to the InfluxDB instance on the management node identifier and the active pipeline configuration baseline or full_cfb. This interval was chosen as a compromise between measurement resolution and monitoring overhead: a shorter period would introduce non-negligible CPU load from the monitoring thread itself, while a longer period would reduce the granularity of the recorded profiles.

A Grafana dashboard provides real-time visualization of the collected metrics across all nodes, comprising two panels: CPU utilization and memory consumption, each filterable by node and pipeline configuration. post-experiment analysis is performed querying the `system_metrics` measurement directly from InfluxDB. Figure 4.16 shows the data flow of the monitoring infrastructure, from metric collection on each node to storage and visualization on the management PC.

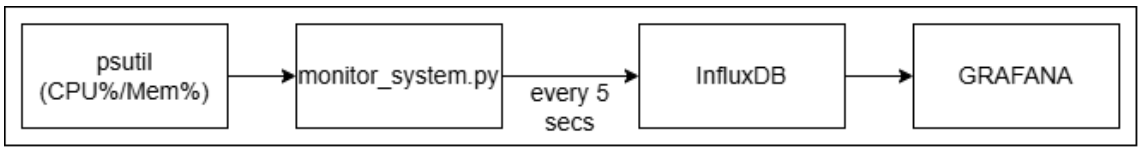


Figure 4.16: Data flow of the monitoring infrastructure, from metric collection on each node to storage in InfluxDB and visualization in Grafana.

This Chapter described the full implementation of the proposed solution, from the hardware and software setup to the extensions introduced in DINASORE and the design of the FBs and CFBs used in the evaluation. The resulting system contributes the experimental platform on which the four test scenarios in Chapter 5 are executed.

Chapter 5

Experimental Evaluation

5.1 Experimental Setup

The three nodes assumed the following roles throughout all test scenarios:

- **Pi 1** (192.168.0.106): edge node for Zone A (150 m^2), running the sensor simulation and HVAC energy calculation pipeline.
- **Pi 2** (192.168.0.103): edge node for Zone B (300 m^2), running an equivalent pipeline with independent parameterization.
- **PC** (localhost): data collection node, running an MQTT subscriber pipeline (*MQTT_SUBSCRIBER_2* connected to INFLUX_DB_2) and hosting InfluxDB and Grafana.

5.2 Evaluation Objectives

The experimental evaluation is structured around three objectives:

1. characterizing the trade-offs introduced by hierarchical CFB composition in terms of computational overhead and observability;
2. evaluating whether memoization-based speculative reuse can mitigate the performance disadvantages associated with CFB encapsulation;
3. deriving practical guidelines for the effective use of CFBs in edge-deployed IEC 61499 applications.

To address objectives 1 and 2, four metrics were collected across all test scenarios:

- **CPU usage (%)**: measured every 5 seconds by the *SystemMonitor* daemon via psutil, representing the computational load imposed by the active pipeline on the edge node.
- **Memory usage (%)**: measured alongside CPU, capturing the memory footprint of the DINASORE runtime and its function blocks.

- **Deploy time (ms)**: the elapsed time from the first FB instantiation to the last, extracted from DINASORE log timestamps, representing the overhead introduced during the deployment phase.
- **OPC UA virtualised nodes**: the number of FB variables exposed as OPC UA nodes, reflecting the observability trade-off introduced by CFB encapsulation.

Objective 3 is addressed in Chapter 6, where the collected results are interpreted and practical recommendations are derived.

5.3 Test Scenarios

Each scenario ran for 30 minutes under steady-state conditions. This duration was chosen to allow *SPECULATIVE_GATE* cache to reach a warm state and for CPU and memory usage to stabilize, while remaining practical for repeated execution across multiple scenarios. All Raspberry Pi nodes were rebooted before each test run to ensure a clean system state with no residual processes or cached state from previous runs.

Four test scenarios were defined, each labeled for reference throughout this chapter:

S1 *single-sensor baseline* - flat FB pipeline, single sensor input, no CFB encapsulation;

S2 *single-sensor full_cfb* - CFB-encapsulated pipeline, single sensor input, with SPECULATIVE_GATE on *Pi2*;

S3 *multi-sensor baseline* - flat FB pipeline, four sensor inputs, no CFB encapsulation;

S4 *multi-sensor full_cfb* - CFB-encapsulated pipeline, four sensor inputs, with SPECULATIVE_GATE.

5.3.1 Single-sensor baseline

The (S1) scenarios uses a flat pipeline of FBs with no CFB encapsulation. Each edge node runs a *SENSOR_SIMULATOR* connected to an *HVAC_ENERGY_CALC* FB, whose output is forwarded to *MQTT_PUBLISHER_2*. The *SENSOR_SIMULATOR* generates temperature readings from a normal distribution centered around 20 at a fixed interval of 1 second. The *HVAC_ENERGY_CALC* FB processes each reading using N_ITER = 20 Newton iterations, introducing a deliberate computational load representative of a real energy calculation workload. The objective of this test is to collect relevant metrics to use as a standard when comparing with future tests.

Results

Table 5.1 summarizes the key metrics recorded during this scenario.

Metric	Pi 1	Pi 2	PC
CPU: initial / final (%)	26,5 / 30	26,5 / 30	—
Memory: initial / final (%)	54,5 / 60	54,5 / 60	—
Deploy time (ms)	1512	2024	981
OPC UA virtualized FBs	3	3	3

Table 5.1: (S1) results.

Figures 5.1 and 5.2 show the CPU and memory usage on *Pi1* and *Pi2* over the 30-minute test window.

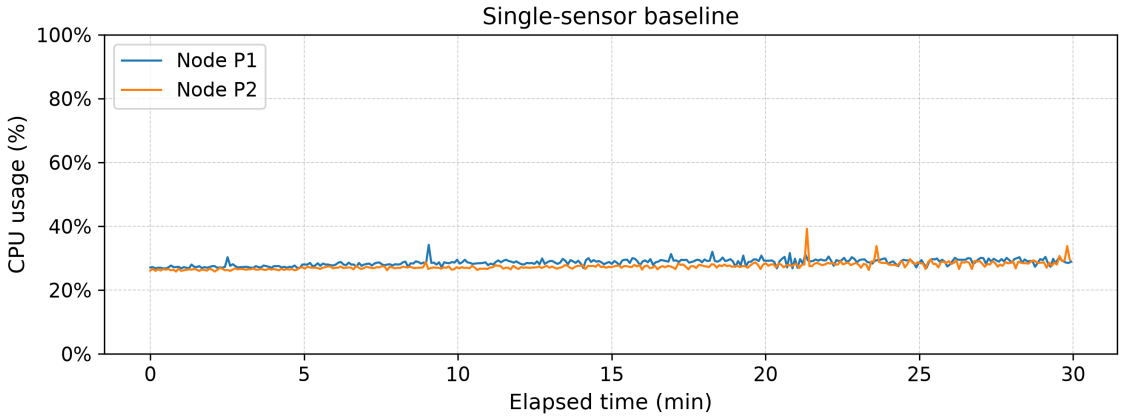


Figure 5.1: CPU usage during the single-baseline test.

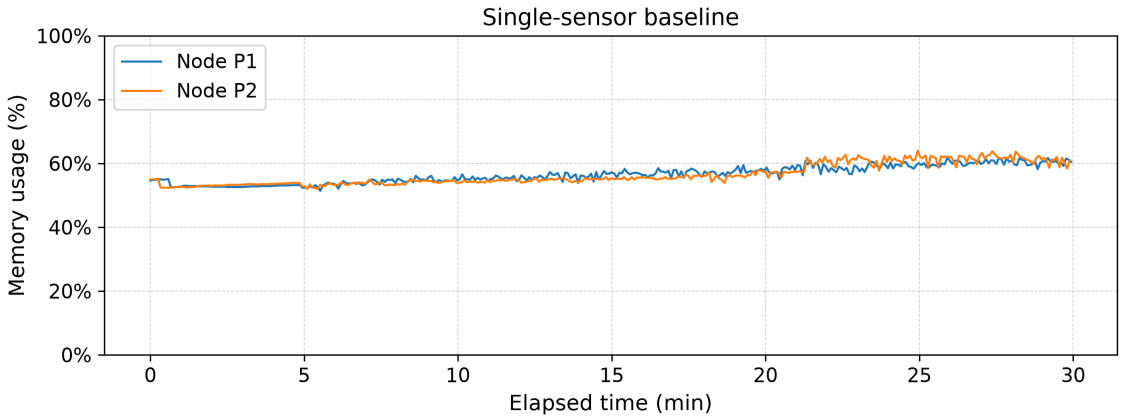


Figure 5.2: Memory usage during the single-baseline test.

5.3.2 Single-sensor full_cfb

The single-sensor full CFB scenario uses the same pipeline as the baseline, but with all FB encapsulated within a CFB. However, the two nodes are configured differently to allow a direct comparison of CFB encapsulation with and without memoization: *Pi1* runs the CFB with-

out a *SPECULATIVE_GATE*, while *Pi2* includes a *SPECULATIVE_GATE* with threshold set to 3, meaning that after 3 consecutive identical input-output pairs, subsequent identical inputs are served from the cache without invoking the *HVAC_ENERGY_CALC* computation. This value was selected as the minimum number of consecutive observations required to establish confidence in a stable input pattern, balancing early cache activation against the risk of caching transient readings, consistent with the design rationale in Section 3.5. The objective of this test is to collect relevant metrics to compare to the previous test to evaluate the effects of CFBs. It also serves as a way to evaluate the performance of the memoization-based FB and its effects on the system as a whole.

Results

Table 5.2 summarizes the key metrics recorded during this scenario.

Metric	**Pi 1**	**Pi 2**	**PC**
CPU: initial / final (%)	26 / 27	26 / 4	—
Memory: initial / final (%)	54,5 / 57	54,5 / 57	—
Deploy time (ms)	1141	1332	971
OPC UA virtualized FBs	1	1	1

Table 5.2: (S2) results.

Figures 5.3 and 5.4 show the CPU and memory usage on *Pi1* and *Pi2* over the 30-minute test window.

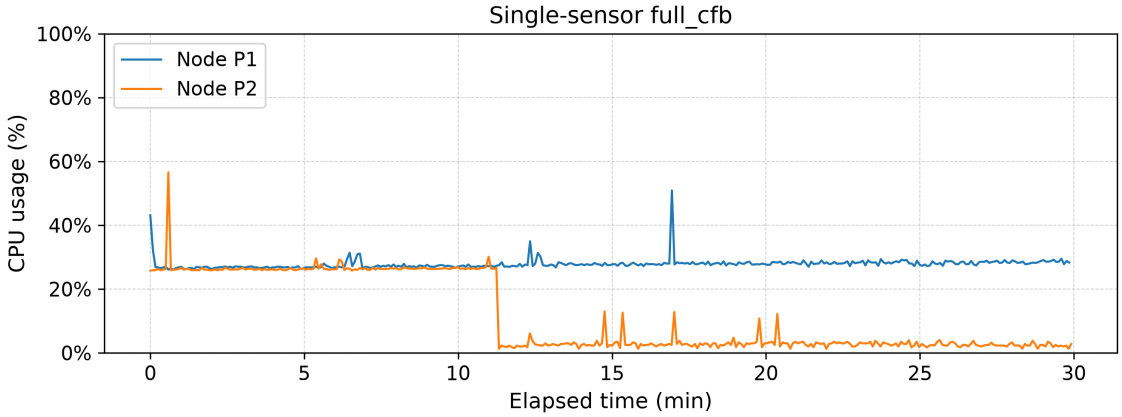


Figure 5.3: CPU usage during the (S2) test.

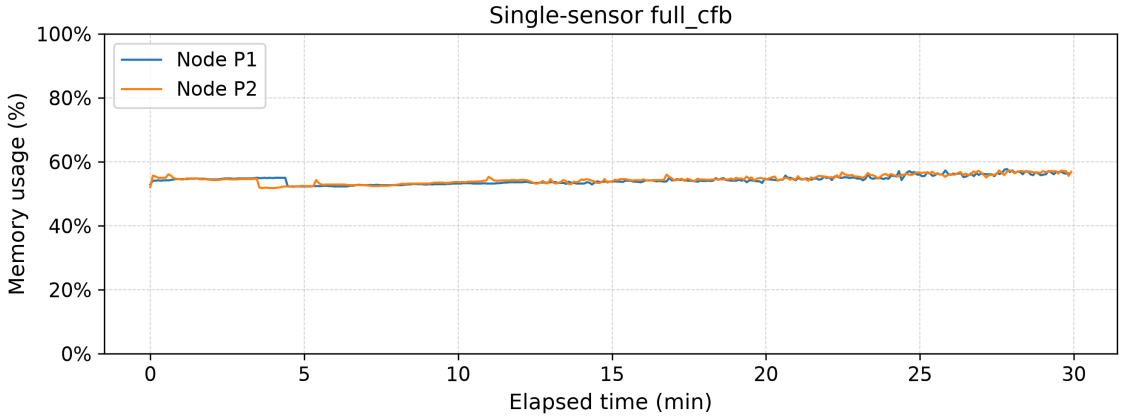


Figure 5.4: Memory usage during the (S2) test.

5.3.3 Multi-sensor baseline

The original test plan included a second edge node (*Pi2*), but this Raspberry Pi had a hardware malfunction that caused under-voltage on its power supply and overheating. This alongside a more CPU intensive test caused the DINASORE to shutdown before the 30 minutes established for all tests. Therefore *Pi2* was removed from the (S3) and (S4) tests and will not be considered when comparing and analyzing this tests.

Furthermore there was an issue with the deploy time of the PC node. On the PC, bytecode compiled during prior sessions ((S1) and (S2)) persists on the disk, causing subsequent deployments to complete significantly faster than the initial run. Since this condition cannot be reproduced consistently across test runs without a full system reinstall, PC deployment time recorded from the multi-sensor tests will not be regarded in the discussion. P1 is rebooted between tests to ensure a clean interpreter state, this solves this issue on the P1 node but it had no effect on the PC node.

The (S3) uses a flat pipeline of FBs with no CFB encapsulation. On *Pi1*, 4 *SENSOR_SIMULATOR_CHAIN* FBs generate temperature readings independently, each cycling through a pre-computed distribution of values derived from a normal distribution centered around 20. Their outputs are aggregated by an AVG FB, which computes the arithmetic mean each cycle. The result is passed to *HVAC_ENERGY_CALC*, which processes each reading using N_ITER = 20 Newton iterations, introducing a deliberate computational load representative of a real energy calculation workload. The objective of this test is to collect data regarding the deployment time in a more complex pipeline with more FBs.

Results

Table 5.3 summarizes the key metrics recorded during this scenario.

Metric	**Pi 1**	**PC**
CPU: initial / final (%)	47 / 52	—
Memory: initial / final (%)	52 / 47	—
Deploy time (ms)	1936	150
OPC UA virtualized FBs	7	2

Table 5.3: (S3) results.

Figures 5.5 and 5.6 show the CPU and memory usage on *Pi1* over the 30-minute test window.

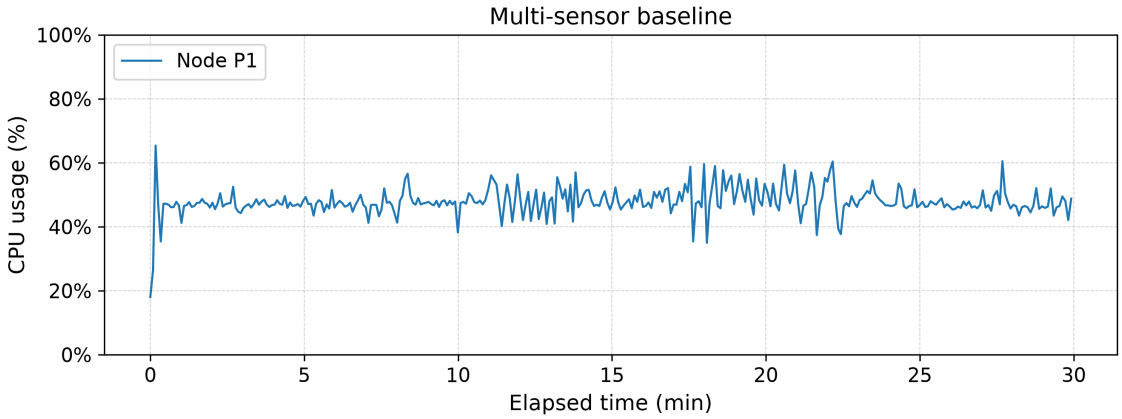


Figure 5.5: CPU usage during the multi-baseline test.

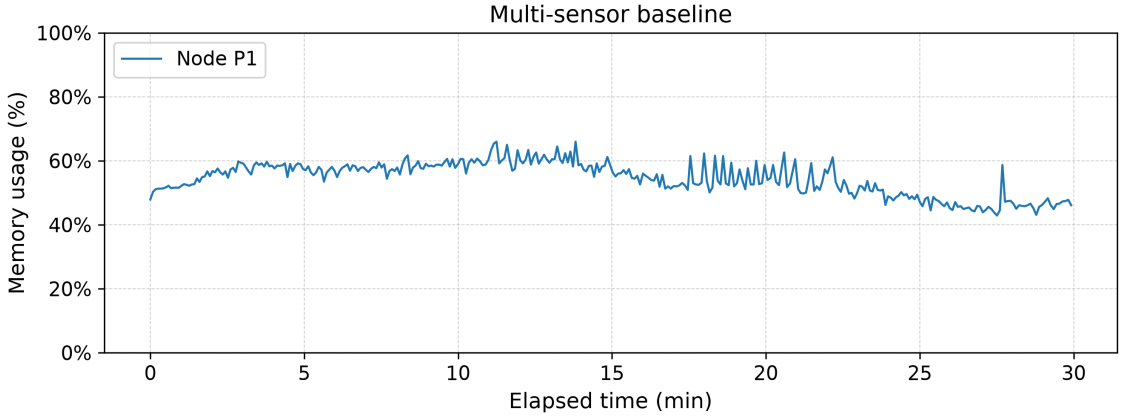


Figure 5.6: Memory usage during the multi-baseline test.

5.3.4 Multi-sensor full_cfb

The (S4) scenario encapsulates the pipeline of the previous test and adds the *SPECULATIVE_GATE* FB to it. This FB receives data from the AVG and *HVAC_ENERGY_CALC* FBs and sends data to the *MQTT_PUBLISHER_2* FB. The objective of this test is to verify how the memoization-based FB will react to this increase in inputs and how the deployment time of the

full_cfb configuration will be affected by this increase in the number of FBs in the internal network.

Results

Table 5.4 summarizes the key metrics recorded during this scenario.

Metric	Pi 1	PC
CPU: initial / final (%)	48 / 50	—
Memory: initial / final (%)	50 / 46	—
Deploy time (ms)	1526	87
OPC UA virtualized FBs	1	1

Table 5.4: (S4) results.

Figures 5.7 and 5.8 show the CPU and memory usage on *Pi1* over the 30-minute test window.

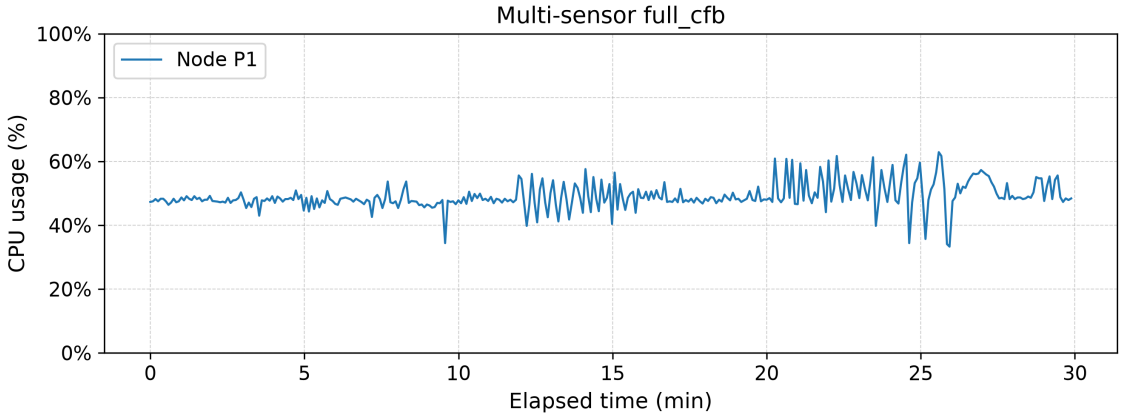


Figure 5.7: CPU usage during the multi-full_cfb test.

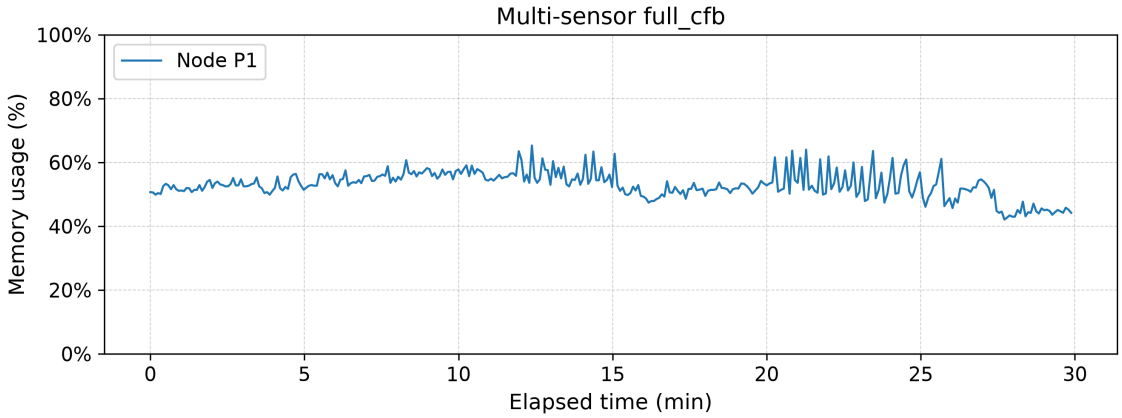


Figure 5.8: Memory usage during the multi-full_cfb test.

The four scenarios presented in this Chapter provide the quantitative basis for the analysis carried out in Chapter 6. Taken together, they cover two levels of pipeline complexity and two configurations, baseline and full_cfb, enabling a direct empirical comparison of the trade-offs introduced by CFB encapsulation and memoization-based speculative reuse.

Chapter 6

Discussion

6.1 Impact of CFBs on Computational Overhead

The (S1) test establishes the performance reference for a flat pipeline of FBs with no CFB encapsulation. On *Pi1* and *Pi2*, CPU usage rose from an initial 26,5% to 30% by the end of the test timer, while memory usage increased from 54,5% to 60%. Deploy time differed slightly between nodes, 1512ms on *Pi1* and 2024ms on *Pi2*, attributable to natural variation in interpreter initialization between the two devices. Each node exposed 3 OPC UA virtualized nodes per FB.

In the (S2) scenario, the most striking result was on *Pi2*: CPU usage fell from 26% at deployment to just 4% at the end of the test, a reduction of 22 percentage points. *Pi1*, which ran the same CFB structure but without a *SPECULATIVE_GATE*, showed no meaningful change, settling at 27%, statistically equivalent to its baseline. This asymmetry between the two nodes is not caused by CFB encapsulation itself, but without a *SPECULATIVE_GATE* present only on *Pi2*, this effect is analyzed in detail in Section 6.2. Memory remained unchanged on both nodes (57%), confirming that CFB encapsulation introduces no additional memory footprint. Deploy time decreased on both nodes: *Pi1* dropped to 1141ms (-24,5%) and *Pi2* to 1332ms (-34,2%) relative to the baseline. This reduction is explained by the deployment mechanism, in the baseline, 4diac issues an individual TCP CREATE request for each FB, accumulating per-FB round trip latency; in the full_cfb scenarios, a single CREATE is issued for the CFB and DINASORE instantiates all internal FBs locally, eliminating that overhead. OPC UA virtualized nodes dropped from 3 to 1 on both nodes, as the internal FB network is hidden behind the CFB interface.

The PC deploy times in the single-sensor tests, 981ms in the (S1) and 971ms in the (S2), are significantly lower than those recorded on either Pi node and show virtually no reduction between scenarios. This is explained by the deployment topology: since 4diac runs on the PC, deployment to the PC is a local operation with no network transmission involved, while deployment to *Pi1* and *Pi2* requires each CREATE request to travel over WiFi via TCP and await acknowledgment. It is precisely this per-FB TCP round trip overhead that CFB encapsulation reduces on the Pi nodes, decreasing deploy time by up to 34%. On the PC, where that overhead is absent, there is no equivalent gain, the 10ms difference between **S1** and (S2) is within measurement noise.

The single-sensor tests show that CFB encapsulation does not increase CPU or memory overhead and actually reduces deploy time. However, these tests involve relatively simple pipelines with 3 FBs per node. To assess wether these effects hold and how they scale when the pipeline is more complex, with a larger number of FBs and a higher event throughput, a second set of tests was conducted with a multi-sensor pipeline. On *Pi1*, 4 *SENSOR_SIMULATOR_CHAIN* FBs run concurrently, each producing readings at approximately 18Hz, whose outputs are aggregated before being passed to *HVAC_ENERGY_CALC*. This represents a more demanding and realistic edge workload, with 7 FBs deployed on *Pi1* in the baseline configuration.

In the (S3) *Pi1* CPU usage rose from 47% to 52% at the end of the test, substantially higher than the (S1), reflecting the combined event load of 4 concurrent sensor simulators. Memory settled at 47%, slightly bellow the initial 52%, a behavior attributed to Python's garbage collector reclaiming memory after initial object allocation. During deployment and FB initialization, Python allocates memory objects, import structures, class definitions and the pre-computed sensor distribution for all 4 *SENSOR_SIMULATOR_CHAIN* instances, that are no longer referenced once the FBs enter steady-state execution; the cyclic garbage collector subsequently identifies and frees these short-lived allocations, bringing memory usage below the initial peak. This contrasts with the single-sensor scenarios, where the lower event rate and smaller initialization footprint mean that the garbage collector pressure was insufficient to offset the gradual accumulation of pipeline state, resulting in memory growing over time rather than decreasing.

In the (S4) scenario, CPU rose from 48% at deployment to 50% at the end of the test, virtually identical to the (S3) and consistent with the pattern observed in the single-sensor tests, CFB encapsulation adds no measurable runtime overhead regardless of pipeline complexity. Memory followed the same downward trend as the baseline, settling at 46% from an initial 50%, again reflecting the garbage collector freeing initialization allocations once steady-state execution is reached. Deploy time fell to 1526ms a reduction of 21,2% relative to the (S3) test, driven by the same mechanism observed in the single-sensor case: a single TCP CREATE replacing seven individual requests. OPC UA virtualized nodes dropped from 7 to 1.

Taken together the four scenarios point to three consistent findings:

1. CFB encapsulation introduces no measurable CPU or memory overhead, the computation performed inside the CFB is identical to the flat pipeline and the dispatch layer adds negligible cost;

2. Deploy time is lower in every full_cfb scenario than its baseline counterpart, in absolute terms the reduction grows with the pipeline complexity (371ms in the single-sensor case versus the 410ms in the multi-sensor case) but in the relative gain decreases (24,5% to 21,2%) as the baseline deploy time itself grows with the number of FBs deployed and the size of the internal FB network of the CFB;

3. The OPC UA observability decreases in proportion to pipeline complexity upon encapsulation, from 3 to 1 virtualized nodes in the single-sensor case and from 7 to 1 in the multi-sensor case;

A closer inspection of the CPU and memory graphs reveals a progressive increase in system instability across the four test scenarios. in the single-sensor tests, both metrics remain largely stable throughout, with minor variability in the final minutes as memory approaches its peak, a consequence of Python's garbage collector triggering progressively larger collection cycles as memory pressure grows. In the (S3), instability is present from the start: four sensors generating events at approximately 18Hz place a continuous load on Python runtime, causing the constant oscillations visible in Figures 5.5 and 5.6. In the (S4) scenario, the pattern is different, CPU begins relatively stable but variability increases progressively over the course of the test, with some moments where it stabilizes again, as visible in Figure 5.7. This temporal degradation reflects the cumulative growth of the *HVAC_ENERGY_CALC* queue: as the backlog of pending computations builds up over time, the competition between queued HVAC calls and incoming sensor events becomes increasingly irregular, producing the the growing instability especially observed towards the end of the test.

These findings carry a practical implication for the industrial adoption of CFBs in distributed IEC 61499 applications: the absence of runtime overhead removes a significant barrier to their use as a structural design pattern. Practitioners can encapsulate pipelines into CFBs for modularity, reuse and deployment efficiency without trading away computational performance. The constraint that emerges is instead at the runtime architecture level, as event throughput increases, the Pyhton execution model imposes a ceiling that is independent of pipeline structure and must be addressed through RTE selection or pipeline-level rate limiting before CFB composition is considered.

6.2 Effectiveness of Memoization-Based Speculative Reuse

The most striking result across all four test scenarios was recorded on *Pi2* during the (S2) test: CPU usage fell from 26% at deployment to 4% by the end of the test, a reduction of 22 percentage points. *Pi1*, which ran an identical CFB structure but without a SPECULATIVE_GATE, remained at 27% throughout, statistically indistinguishable from its baseline value. This contrast isolates the effect of memoization: the CFB structure alone does not reduce CPU and the reduction on *Pi2* is attributable entirely to the *SPECULATIVE_GATE* bypassing *HVAC_ENERGY_CALC* on cache hits.

The CPU drop on *Pi2* was not immediate. The reduction became visible approximately 11 to 12 minutes into the test, corresponding to the cache warm up phase. With *WARMUP_COUNT* set to 3, the *SPECULATIVE_GATE* requires 3 confirmed input-output pairs per cache key before serving results from cache. Since the *SENSOR_SIMULATOR* generates values drawn from a normal distribution centered around 20ºC at 1Hz, the same temperature values recur frequently. Over the first minutes of the test, each distinct value accumulates its three required hits; once the cache is sufficiently warm, the vast majority of incoming values match a cached entry and *HVAC_ENERGY_CALC* is no longer invoked, causing CPU to drop sharply and stabilize at 4%.

In the (S4) scenario, *Pi1* also ran a *SPECULATIVE_GATE*, but the effect on CPU was negligible: usage settled at 50%, compared to 52% in the (S3), a difference of 2 percentage points

falls within the variability observed throughout the multi-sensor tests. Two factors explain why memoization failed to produce a meaningful reduction here. First, the high event rate generated by four concurrent *SENSOR_SIMULATOR_CHAIN* FBs at approximately 18Hz meant that *HVAC_ENERGY_CALC* received new requests faster than it could process them, causing a growing queue of pending computations. Even during periods of cache hits, the HVAC was still running to drain that backlog, preventing CPU from dropping. Second, the AVG FB produced outputs with a large number of decimal places, introducing a wider variety of cache keys than the single-sensor scenario. As a result, the cache could not stabilize and cover the full key space within the 30-minute test window, limiting the proportion of requests served from cache.

These results indicate that memoization-based speculative reuse is effective when the cached computation is the dominant contributor to CPU load, the event rate is low enough to prevent queue buildup and the input value space is sufficiently constrained for the cache warm up within a realistic operating window, conditions that were all satisfied in the single-sensor scenario, where one sensor operating at 1Hz fed a heavy deterministic computation with a limited and repeating set of input values.

However, the technique is highly sensitive to increase in pipeline complexity that introduce non-deterministic behavior. The single-sensor pipeline is inherently deterministic: one sensor produces one value per cycle, the computation is always invoked with that value and the output is stable and reproducible. Adding four sensors organized in a chain, where each sensor triggers the next, changes this fundamentally: the AVG FB aggregates readings that vary across the chain, producing a wider and less predictable distribution of output values. This additional variability is sufficient to prevent the cache from stabilizing within the test window and the high event rate compounds the problem by creating a queue of pending computations that persists even during cache hits. What was originally a highly cacheable workload becomes resistant to memoization not because the technique is flawed, but because the assumptions it relies upon, stable, repeating, low-frequency inputs, no longer hold. Memoization-based speculative reuse is therefore best suited to pipelines where determinism is preserved end-to-end and should be applied with caution as pipeline complexity grows.

This sensitivity to pipeline determinism has a direct implication for how speculative reuse should be positioned in the IEC 61499 ecosystem. The *SPECULATIVE_GATE* represents a specialized mechanism for a specific class of workloads, computation-heavy, low-frequency, deterministic pipelines, a profile that is common at the edge of a CPPS, where individual sensors feed isolated control loops. As pipelines aggregate data across multiple sensors or implement higher-level coordination logic, the determinism that makes memoization effective is progressively eroded. Future IEC 61499 applications that aim to combine data aggregation with speculative optimization would require architectural provisions, such as a normalization or input filtering stages, to preserve the input stability that the technique relies upon.

6.3 CPU, Memory and Observability Trade-offs

6.3.1 Encapsulation vs Observability

The reduction in OPC UA virtualized nodes observed across the full_cfb scenarios represents the most operationally significant trade-off introduced by CFB encapsulation. In the baseline scenarios each FB exposes its variables independently, allowing an operator to monitor individual sensor readings, HVAC inputs and outputs and MQTT publisher state in real time through any OPC UA client. Once those FBs are encapsulated within a CFB, only the CFB's own interface remains visible, the internal network is entirely hidden from the OPC UA layer. In multi-sensor scenario, this means that six variables that were previously accessible, the four individual sensor readings, the averaged output and the HVAC result, become invisible to external monitoring tools. This loss of observability has direct practical consequences. diagnosing a misbehaving sensor or an anomalous HVAC output requires either re-deploying the pipeline in its flat form or adding explicit diagnostic outputs to the CFB interface, neither of which is trivial in a running edge deployment. The encapsulation benefits, simpler deployment, reduced TCP overhead and cleaner abstraction, come at the cost of reduced runtime introspection and this trade-off becomes more pronounced as the number of FBs inside the CFB grows.

6.3.2 CPU vs Memory

The *SPECULATIVE_GATE* introduces a theoretical trade-off between CPU and memory: computed results are stored in a in-memory cache and as the number of distinct input values grows, so does the cache footprint. However, this trade-off could not be isolated or quantified from the data collected in these experiments. The memory differences observed between baseline and full_cfb scenarios are explained by the garbage collector dynamics discussed in Section 6.1. The cache itself, storing a few dozen float key value pairs, is negligible relative to the total memory footprint of the Python runtime. While the trade-off remains relevant at scale, particularly on memory-constrained devices such as the Raspberry Pi, these experiments were not designed to isolate it and any claim about its magnitude would not be supported by the collected data.

6.3.3 Determinism vs Throughput

The third trade-off is between pipeline determinism and throughput. The single-sensor pipeline is highly deterministic: one sensor produces one value per second, the computation is always invoked with that value and the output is stable and reproducible across cycles. This determinism is precisely what makes memoization effective, repeated inputs reliably produce cache hits and the cache stabilizes quickly. increasing throughput by adding sensors organized in a chain disrupts this property. The AVG FB aggregates values that vary independently across the chain, producing a distribution of outputs that is wider and less predictable than any individual sensor. The higher event rate further compounds this by introducing timing variability and queue buildup in the HVAC computation. The result, as observed in the multi-sensor tests, is a system where the

conditions for effective memoization, stable, repeating, low-frequency inputs, no longer hold. This reveals a fundamental tension in IEC 61499 edge deployments: increasing the number of sensor or the event throughput of a pipeline, which is generally desirable for richer data collection, directly undermines the determinism that speculative reuse relies upon. The two objectives are not easily reconciled without architectural changes that explicitly preserve input stability, such as additional normalization or filtering stages before the *SPECULATIVE_GATE*.

6.3.4 Modularity, Reuse and Scalability

The quantitative metrics in Chapter 55 characterize the runtime and deployment cost of CFB encapsulation but do not fully capture its structural benefits. A qualitative assessment of modularity, reuse and scalability provides a more complete picture of what hierarchical composition offers in practice.

In terms of reuse, the CFB model demonstrated in this thesis allows the same pipeline definition to be instantiated on multiple nodes without modification. *PIPELINE_P1* type was deployed independently to *Pi1* and *Pi2*, each node executing its own instance of the enclosed FB network. In a flat pipeline configuration, the same effects would require managing individual FB deployments on each node separately; the CFB packages the entire pipeline into a single reusable unit, so any structural change to the pipeline definition propagates to all nodes that instantiate it. This reuse benefit becomes increasingly relevant as the number of nodes in a distributed application grows.

Modularity is reinforced by the encapsulation boundary itself. The internal FB network is hidden behind the CFB interface, creating a separation between a pipeline's internal implementation and its external contract. Changes to the internal structure, such as inserting the *SPECULATIVE_GATE*, do not require modifying the CFB interface or any external configuration as long as the interface remains compatible. This separation simplifies reasoning about system behavior at the deployment and reduces the risk of unintended side effects when the internal pipeline evolves.

From a scalability perspective, CFB encapsulation reduces the visible model complexity as pipeline size grows. A flat seven-FB pipeline exposes seven individual types, seven OPC UA endpoints and seven deployment entries; the equivalent CFB-based pipeline reduces this to a single deployable unit with one external interface. As applications scale to more FBs or nodes, the operational overhead of managing individual FB deployments grows linearly in the flat case but remains bounded by the CFB interface in the encapsulated case. The experiments suggest, however, that scalability remains constrained by the Python runtime independently of pipeline structure, as the queue buildup in the multi-sensor scenario demonstrates.

6.4 Limitations and Threats to Validity

The most significant structural limitation of this evaluation was the hardware malfunction on *Pi2*, which caused it to shut down during multi-sensor tests due to under-voltage and overheating. This restricted the multi-sensor scenarios to *Pi1* only, reducing the comparative scope of those

tests. The impact of this limitation is partially mitigated by the behavior observed in the single-sensor tests, where both nodes ran identical pipelines and produced equivalent metrics, CPU and memory were consistently aligned between the two nodes when the pipeline configuration was the same. *Pi2* was intended as a redundant node but as a controlled point of comparison: its purpose was specifically to run a pipeline without a *SPECULATIVE_GATE* alongside *Pi1*, which ran the same pipeline, enabling a direct isolation of the memoization effect. The conclusions drawn from the (S2) test therefore remain in the scope, but the absence of *Pi2* in the multi-sensor tests limits the ability to replicate the same controlled, real time comparison at higher pipeline complexity.

The use of simulated rather than physical sensors introduces a further threat to external validity. The *SENSOR_SIMULATOR* generates values from a pre-computed normal distribution, producing a controlled and repeating input sequence. Real sensors introduce noise, drift and occasional outliers that would affect the distribution of cache keys in the *SPECULATIVE_GATE* and likely reduce its hit rate relative to what was observed here. The results reported for memoization effectiveness should therefore be interpreted as an upper bound performance under idealized input conditions.

Network variability over WiFi represents a further source of uncertainty in the deploy time measurements. All communication between the PC running 4diac and the Raspberry Pi nodes traveled over a local WiFi network, where transmission delays are not constant. While the deploy time measurements were taken from DINASORE log timestamps rather than from the network layer, any queuing or retransmission at the TCP level would be captured in those measurements, introducing variability that cannot be attributed solely to the runtime behavior of the nodes.

The *SystemMonitor* daemon sampled CPU and memory every 5 seconds, which limits the temporal resolution of the collected metrics. Short-duration spikes or transient load changes lasting less than 5 seconds may be missed or averaged out in the recorded values, meaning that the initial and final percentages reported in the result tables represent sampled snapshots rather than precise instantaneous measurements.

The results are specific to the Python implementation of DINASORE and are influenced by the Python Global Interpreter Lock [50], which serializes thread execution and shapes the concurrency behavior observed under multi-sensor workloads. Extrapolating these findings to other IEC 61499 RTEs requires distinguishing between effects that are protocol-level and those driven by Python runtime characteristics. The deploy time reductions are likely generalized: CFB encapsulation replaces multiple individual TCP CREATE requests with a single one regardless of the RTE implementation language, making this a property of the 4diac deployment protocol rather than than DINASORE's specifically. The OPC UA observability reduction follows from the IEC 61499 specification itself and would apply to any compliant RTE. the CPU findings however, are more implementation-specific. RTEs such as FORTE execute without a Global Interpreter Lock [50], enabling true parallel processing; the queue buildup observed under multi-sensor workloads would likely be less severe in such environments. The *SPECULATIVE_GATE* cache would also require explicit synchronization in a multi-threaded C++ RTE, introducing overhead not present

in the Python version and potentially reducing the net CPU saving from memoization. The memoization results reported here should therefore be treated as specific to the Python execution model.

The *WARMUP_COUNT* threshold of 3 was constant in all tests. A higher threshold would delay cache activation and reduce the warm up-phase CPU drop, while a lower threshold would risk caching results before sufficient convergence, affecting the accuracy of cached outputs.

Lastly, the scenario space is limited. Only two levels of pipeline complexity were evaluated, three and seven FBs per node, with no intermediate configurations, preventing a continuous characterization of how deploy time scaling, CPU overhead and memoization effectiveness evolve as the pipeline grows. The evaluation was furthermore conducted on a homogeneous two-node of identical Raspberry Pi devices. Testing across a greater number of nodes and more varied hardware configurations would produce more generalizable conclusions and allow the reported trade-offs to be validated across a wider range of deployment scenarios.

6.5 Design Guidelines for CFB-Based CPS

The results collected across the four tests scenarios support a set of practical guidelines for the design of CFB-based IEC 61499 applications on constrained edge hardware and CPS.

Guideline 1: Use CFBs as the primary structuring mechanism in distributed deployments over high-latency links.

In distributed IEC 61499 deployments where nodes communicate over WiFi or other high-latency links, the per-FB TCP round-trip latency accumulates during flat pipeline complexity and grows linearly with the number of FBs deployed. CFB encapsulation should be the preferred structuring mechanism whenever deployment time is a concern. By replacing multiple individual CREATE requests with a single one, CFB encapsulation eliminates this overhead entirely. The absolute saving scales with pipeline complexity. The number of OPC UA observable nodes decreases proportionally with the number of FBs enclosed, a cost that must be weighed against the deployment benefit at design time.

Guideline 2: Use CFBs freely for modularity and abstraction without concern for runtime overhead.

Developers on resource-constrained edge hardware may be reluctant to introduce CFB encapsulation if they suspect it adds computational overhead at runtime. CFBs should be applied freely for modularity, reusability and abstraction without concern for runtime performance penalties. CPU and memory usage were statistically indistinguishable between flat and CFB-encapsulated pipelines across all scenarios, confirming that the dispatch layer introduced by the CFB boundary is negligible relative to the computation performed by the enclosed FBs. The main cost of encapsulation is structural rather than computational: CFB interfaces require careful design upfront, since a poorly defined interface that exposes insufficient outputs reduces runtime observability.

Guideline 3: Design CFB interfaces with observability requirements in mind before deployment.

CFB encapsulation hides the internal FB network from the OPC UA monitoring layer, reducing observable nodes from three to one in the single sensor case and from seven to one in the multi-sensor case. All variables that need to be accessible at runtime must be explicitly exposed through the CFB output interface at design time. Any variable not included in the interface becomes inaccessible to OPC UA clients without redeployment and retrofitting observability into a running edge system is operationally costly. Exposing additional internal variables increases the CFB interface size and requires more upfront design effort, partially offsetting the simplification that encapsulation provides.

Guideline 4: Apply memoization-based speculative reuse selectively to stable, deterministic, computational-heavy pipelines.

Memoization via *SPECULATIVE_GATE* requires three conditions to be effective: the cached computation must dominate CPU load, the event rate must be low enough to prevent a backlog of pending computations and the input value space must be constrained enough for the cache to stabilize within the operational window. The *SPECULATIVE_GATE* should be treated as a targeted optimization for pipelines where these conditions are verifiably met, rather than a general-purpose technique. In the single-sensor scenario, where all three conditions held, CPU usage on *Pi2* fell from 26% to 4% once the cache warmed up. In the multi-sensor scenario, the higher event rate and wider input distribution prevented stabilization and no measurable reduction was observed. When the required conditions do not hold, the *SPECULATIVE_GATE* adds architectural complexity without computational benefits and risks aggravating queue instability.

6.6 Validation of the Research Hypothesis

This section answers **RQ1**, **RQ2** and **RQ3** individually and then assesses wether the results collectively support **H1**.

RQ1 - Runtime overhead of CFB encapsulation.

CPU and memory usage were statistically indistinguishable between **S1** and **S2** and **S3** and **S4**, across al tested pipeline configurations. The dispatch layer introduced by the CFB boundary adds negligible overhead relative to the computation performed by the enclosed FBs. **RQ1 is answered: CFB encapsulation introduces no measurable runtime overhead.**

RQ2 - Effect on deployment characteristics, modularity and maintainability.

Deployment time decreased in every full_cfb scenario: replacing multiple TCP CREATE requests with a single CFB CREATE eliminates per-FB round-trip latency over WiFi links. The architectural cost is a reduction in OPC UA observability, proportional to the number of enclosed FBs, from three observable nodes to one in **S2** and from seven to one in **S4**. This trade-off must be managed at design time by explicitly exposing variables that require runtime monitoring. **RQ2 is answered: CFBs reduce deployment time and improve modularity, at the cost of reduced OPC UA observability.**

RQ2 - memoization-based speculative reuse as a compensatory mechanism.

Under deterministic, slowly varying workloads, the *SPECULATIVE_GATE* yielded substantial CPU reductions, as observed in **S2** on *Pi2*. Under increased pipeline complexity (**S4**), queue instability emerged and the mechanism provided no benefit. **RQ3 is answered: memoization-based speculative reuse is effective under deterministic, low-frequency workloads but degrades as pipeline complexity and event throughput increase.**

Validation of H1.

The results collectively support **H1**: CFB encapsulation improved modularity and reduced deployment overhead without introducing measurable runtime cost, on a Python-based RTE deployed on a resource-constrained hardware. The conditions under which this holds are well-characterized: the technique is computationally free but architecturally consequential and any embedded optimization must be matched to the determinism properties of the enclosed pipeline.

Chapter 7

Conclusion

Deploying intelligent control logic across resource-constrained edge devices in industrial CPSs demands software abstractions that are both expressive and operationally lightweight. IEC 61499 CFBs offer a natural candidate for this role, enabling the encapsulation of complex multi-FB pipelines into single deployable units, but the practical cost of this abstraction on real constrained hardware had not been empirically established. This thesis addressed that gap through the design, implementation and evaluation of CFB-based execution in DINASORE, deployed o Raspberry Pi hardware had assessed four experimental scenarios. The results provided a grounded characterization of what CFB encapsulation costs, what it saves, and under which conditions its associated optimization techniques remain effective.

This thesis investigated the use of IEC 61499 CFBs as a structuring mechanism in distributed CPS pipelines deployed on resource constrained edge hardware and empirically characterized the trade-offs of their adoption in terms of computational overhead, deployment time and runtime observability.

The software contributions of this thesis are twofold. First, the DINASORE RTE was extended to support CFB-based deployments, involving the design and implementation of a CFB execution engine that enables encapsulation of FBs into a single deployable unit via 4diac. Second, a memoization-based speculative reuse FB, the *SPECULATIVE_GATE*, was designed and integrated within the CFB structure, providing a reusable building block for computational optimization in IEC 61499 pipelines.

The scientific contribution of this thesis is the empirical characterization of the trade-offs introduced by CFB encapsulation in distributed IEC 61499 deployments on resource-constrained edge hardware. The evaluation established that CFB encapsulation introduces no measurable CPU or memory overhead while consistently reducing deployment time. It further established that memoization-based speculative reuse is effective under deterministic, low-frequency workloads but degrades as pipeline complexity and event throughput increase. From these findings, practical desig guidelines were derived for the effective use of CFBs in edge deployments.

The central insight this thesis contributes is that CFB encapsulation is not merely a structural convenience, it is also a computational enabler. The CFB boundary that hides internal FBs from

the deployment layer also provides a natural side for transparent optimization: by intercepting inputs before they reach computationally intensive stages, the *SPECULATIVE_GATE* demonstrates that CFBs can reduce CPU load without any modifications to the enclosed logic or the external application. Before this work, the runtime overhead of hierarchical composition on constrained edge hardware was an open empirical question; the results presented here establish that CFB encapsulation adds no measurable CPU or memory costs regardless of pipeline complexity, while simultaneously enabling deployment-transparent optimization. The trade-offs that matter are instead architectural: reduced OPC UA observability proportional to encapsulated pipeline depth and sensitivity of in-CFB optimization to the determinism properties of the enclosed workload.

7.1 Limitations of the Proposed Solution

While this thesis demonstrated the viability of CFB-based deployments and their optimal usage scenarios, it also demonstrated several of the limitations this proposed solutions possesses in its broader applicability.

The *SPECULATIVE_GATE* lacks a rate-limiting mechanism which is needed when the event rate in the pipeline exceeds the processing capacities of the cached computation, which results in an increasing queue that grows uncontrollably. As observed in the tests, this queue buildup causes the CPU to remain saturated even in periods of multiple consecutive cache hits, negating all benefits of the memoization mechanism. Any implementation of this technique in a IEC 61499 CPS would require some mechanism capable of reducing or limiting this queue when the event rate exceeds a threshold.

The current version of DINASORE exposes only the CFB interface to the OPC UA layer, hiding its FB internal network from external monitoring tools. To restore observability a developer must take in consideration the outputs of the CFB interface during the design of the CFB. As monitoring needs evolve the CFB interface can be changed, allowing the developer access to internal variables at the end of each internal cycle resolution. This limitation is manageable during design, but it's still a limitation that any developer that decides to use CFBs must be aware.

Lastly the evaluation of this proposed solutions should have had larger and more varied testbed in order to strengthen the generality of the conclusions and guidelines. testing CFB-based deployments using more nodes, physical sensors or simply a more varied set of hardware would result in more valid conclusions and would prove if the trade-offs concluded could hold in a great variety of scenarios.

7.2 Future Work

The conclusions and limitations of this thesis open several directions of future investigative work within the CFB-based distributed CPS.

This thesis explored memoization as one technique to reduce computational load within CFB-based pipelines. An interesting follow up would be an investigation on other optimization techniques and mechanisms that would be compatible with CFBs and IEC 61499 execution model. Approaches such as lazy evaluation, predictive pre-computing based on historical inputs or adaptive scheduling of FB execution as alternatives or complements to memoization.

This thesis also raises questions related to the observability and encapsulation in CFBs. The questions regarding monitoring architectures within IEC 61499 systems become relevant for this type of situations. Future works could investigate a selective observability mechanism for an CFBs internal FB network or explore how existing industrial monitoring systems standards could be extended to support hierarchical FB introspection.

The evaluation tests conducted in this thesis, as its explained in Section 7.1, could be broader to test if the conclusions of this thesis are valid with physical sensors or other types of constrained hardware besides Raspberry Pis. Further test could verify if other communication protocols have any benefits for CFB deployments or if RTEs implemented in other languages have any effect in CPU or memory usage.

Lastly this thesis defined the CFB internal FB network manually, deciding which FBs to group together and how to partition the pipeline across the nodes based on the developers judgment. This was possible because the pipelines that were tested were relatively simple and straightforward, but what about cases were we observe very complex pipelines with a high number of FBs distributed between various devices? Future works could study whether a automated approach to CFB partition is feasible, exploring how to optimally group FBs into CFBs in large scale CPS distributed between various nodes. This would use metrics defined by this thesis such as deploy time, maximizing observability or balancing computational load between nodes. This would turn CFB-based design from a manual and specialized activity to a more principled engineering process, particularly relevant as CPS complexity and the number of deployed nodes grow.

References

[1] H. Lasi, P. Fettke, H.-G. Kemper, T. Feld, and M. Hoffmann, "Industry 4.0," *Business & information systems engineering*, vol. 6, no. 4, pp. 239–242, 2014.

[2] R. Rajkumar, I. Lee, L. Sha, and J. Stankovic, "Cyber-physical systems: The next computing revolution," in *Proceedings of the 47th design automation conference*, 2010, pp. 731–736.

[3] S. J. Oks et al., "Cyber-physical systems in the context of industry 4.0: A review, categorization and outlook," *Information Systems Frontiers*, vol. 26, no. 5, pp. 1731–1772, 2024.

[4] G. Lyu and R. W. Brennan, "Towards iec 61499-based distributed intelligent automation: A literature review," *IEEE Transactions on Industrial Informatics*, vol. 17, no. 4, pp. 2295–2306, 2020.

[5] V. Ashiwal, O. Miguel-Escrig, B. Wiesmayr, A. Zoitl, and J.-A. Romero-Pérez, "Identification and evaluation of pitfalls in the migration from iec 61131-3 to iec 61499: A review," *IEEE Open Journal of the Industrial Electronics Society*, 2025.

[6] G. Lyu and R. W. Brennan, "Multi-agent modelling of cyber-physical systems for iec 61499-based distributed intelligent automation," *International Journal of Computer Integrated Manufacturing*, vol. 38, no. 5, pp. 596–622, 2025.

[7] A. Zoitl, T. Strasser, and G. Ebenhofer, "Developing modular reusable iec 61499 control applications with 4diac," in *2013 11th IEEE International Conference on Industrial Informatics (INDIN)*, IEEE, 2013, pp. 358–363.

[8] J. H. Christensen, T. Strasser, A. Valentini, V. Vyatkin, and A. Zoitl, "The iec 61499 function block standard: Overview of the second edition," *ISA autom. Week*, vol. 6, pp. 6–7, 2012.

[9] A. Hopsu, U. D. Atmojo, and V. Vyatkin, "On portability of iec 61499 compliant structures and systems," in *2019 IEEE 28th International Symposium on Industrial Electronics (ISIE)*, IEEE, 2019, pp. 1306–1311.

[10] L. Sonnleithner, B. Wiesmayr, V. Ashiwal, and A. Zoitl, "Iec 61499 distributed design patterns," in *2021 26th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, IEEE, 2021, pp. 1–8.

[11] B. Wiesmayr, A. Zoitl, A. Garmendia, and M. Wimmer, "A model-based execution framework for interpreting control software," in *2021 26th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, IEEE, 2021, pp. 1–8.

[12] S. Gao, K. Qin, X. Zhang, and W. Dai, "Prioritized deterministic real-time execution semantics for industrial edge applications based on iec 61499 event types," *IEEE Transactions on Industrial Informatics*, 2025.

[13] W. Dai et al., "Concurrent deterministic execution semantics for iec 61499-based ot-it convergence industrial edge applications," *IEEE Open Journal of the Industrial Electronics Society*, 2025.

[14] T. Strasser, A. Zoitl, J. H. Christensen, and C. Sünder, "Design and execution issues in iec 61499 distributed automation and control systems," *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, vol. 41, no. 1, pp. 41–51, 2010.

[15] M. Sadeghi, "Developing iec61499 in industrial processes, measurement and control systems (ipmcs)," *WSEAS Transactions on Systems and Control*, vol. 5, no. 04, 2010.

[16] M. Xavier, S. Patil, and V. Vyatkin, "Cyber-physical automation systems modelling with iec 61499 for their formal verification," in *2021 IEEE 19th International Conference on Industrial Informatics (INDIN)*, IEEE, 2021, pp. 1–6.

[17] V. Shatrov and V. Vyatkin, "Promela formal modelling and verification of iec 61499 systems with comparison to smv," in *2021 IEEE 19th International Conference on Industrial Informatics (INDIN)*, IEEE, 2021, pp. 1–6.

[18] W. Dai, Y. Zhang, Y. Zhang, J. Kang, and D. Huang, "Automatic information model generation for industrial edge applications based on iec 61499 and opc ua," *IEEE Transactions on Industrial Informatics*, vol. 19, no. 4, pp. 6093–6104, 2022.

[19] D. Drozdov, V. Dubinin, S. Patil, and V. Vyatkin, "A formal model of iec 61499-based industrial automation architecture supporting time-aware computations," *IEEE Open Journal of the Industrial Electronics Society*, vol. 2, pp. 169–183, 2021.

[20] V. Vyatkin, "Iec 61499 as enabler of distributed and intelligent automation: State-of-the-art review," *IEEE transactions on Industrial Informatics*, vol. 7, no. 4, pp. 768–781, 2011.

[21] K. Goh, W. Ding, and B. Tjahjono, "Iterative knowledge based code generator for iec 61499 function block," in *TENCON 2009-2009 IEEE Region 10 Conference*, IEEE, 2009, pp. 1–6.

[22] N. Tapoglou, J. Mehnen, M. Doukas, and D. Mourtzis, "Optimal machining parameter selection based on real-time machine monitoring using iec 61499 function blocks for use in a cloud manufacturing environment: A case study for face milling," in *International Manufacturing Science and Engineering Conference*, American Society of Mechanical Engineers, vol. 45806, 2014, V001T04A015.

[23] T. Liakh, R. Sorokin, D. Akifev, S. Patil, and V. Vyatkin, "Formal model of iec 61499 execution trace in fbme ide," in *2022 IEEE 20th International Conference on Industrial Informatics (INDIN)*, IEEE, 2022, pp. 588–593.

[24] C. Sunder, A. Zoitl, J. H. Christensen, M. Colla, and T. Strasser, "Execution models for the iec 61499 elements composite function block and subapplication," in *2007 5th IEEE International Conference on Industrial Informatics*, IEEE, vol. 2, 2007, pp. 1169–1175.

[25] I. Hegny, T. Strasser, M. Melik-Merkumians, M. Wenger, and A. Zoitl, "Towards an increased reusability of distributed control applications modeled in iec 61499," in *Proceedings of 2012 IEEE 17th International Conference on Emerging Technologies & Factory Automation (ETFA 2012)*, IEEE, 2012, pp. 1–8.

[26] B. Wiesmayr, S. Mehlhop, and A. Zoitl, "Close enough? criteria for sufficient simulations of iec 61499 models," in *2023 IEEE 19th International Conference on Automation Science and Engineering (CASE)*, IEEE, 2023, pp. 1–7.

[27] A. Zoitl, T. Strasser, and A. Valentini, "Open source initiatives as basis for the establishment of new technologies in industrial automation: 4diac a case study," in *2010 IEEE International Symposium on Industrial Electronics*, IEEE, 2010, pp. 3817–3819.

[28] M. Huang, X. Wu, and W. Dai, "Using dynamic interface function block in iec 61499-based industrial edge applications," in *2022 IEEE 17th Conference on Industrial Electronics and Applications (ICIEA)*, IEEE, 2022, pp. 845–850.

[29] G. Pinto, A. Nunes, L. Silva, R. Pinto, J. Pinheiro, and A. Ribeiro, "Digital factory: An industrial case study for function block-oriented development," in *2023 IEEE 9th World Forum on Internet of Things (WF-IoT)*, IEEE, 2023, pp. 01–06.

[30] U. OPC, *Opc unified architecture*, 2008.

[31] J. T. Da Silva, A. L. Dias, and I. N. Da Silva, "A survey on opc ua protocol: Overview, challenges and opportunities," in *2023 15th IEEE International Conference on Industry Applications (INDUSCON)*, IEEE, 2023, pp. 1523–1530.

[32] B. Wiesmayr, A. Zoitl, and R. Rabiser, "Assessing the usefulness of a visual programming ide for large-scale automation software," in *2021 ACM/IEEE 24th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, IEEE, 2021, pp. 297–307.

[33] R. Sorokin, S. Patil, and V. Vyatkin, "Novel development tool for iec 61499 based on domain-specific languages," *IFAC-PapersOnLine*, vol. 55, no. 2, pp. 439–444, 2022.

[34] L. Kong et al., "Edge-computing-driven internet of things: A survey," *ACM computing surveys*, vol. 55, no. 8, pp. 1–41, 2022.

[35] M. S. Ramanujam, "Leveraging structural stability for efficient compute-memory tradeoffs in edge systems," Ph.D. dissertation, Princeton University, 2025.

[36] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," *IEEE internet of things journal*, vol. 3, no. 5, pp. 637–646, 2016.

[37] C.-H. Hong and B. Varghese, "Resource management in fog/edge computing: A survey on architectures, infrastructure, and algorithms," *ACM computing surveys (csur)*, vol. 52, no. 5, pp. 1–37, 2019.

[38] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, "Fog computing and its role in the internet of things," in *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*, 2012, pp. 13–16.

[39] A. Estebanez, D. R. Llanos, and A. Gonzalez-Escribano, "A survey on thread-level speculation techniques," *ACM Computing Surveys (CSUR)*, vol. 49, no. 2, pp. 1–39, 2016.

[40] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. Elsevier, 2011.

[41] E. M. Pereira, J. P. C. dos Reis, and G. Gonçalves, "Dinasore: A dynamic intelligent reconfiguration tool for cyber-physical production systems.," in *SAM IoT*, 2020, pp. 63–71.

[42] B. Mishra and A. Kertesz, "The use of mqtt in m2m and iot systems: A survey," *Ieee Access*, vol. 8, pp. 201 071–201 086, 2020.

[43] I. Brumar, M. Casas, M. Moreto, M. Valero, and G. S. Sohi, "Atm: Approximate task memoization in the runtime system," in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, IEEE, 2017, pp. 1140–1150.

[44] Raspberry Pi Foundation, *Raspberry Pi 3 model B*, https://www.raspberrypi.com/products/raspberry-pi-3-model-b/, 2016.

[45] A. Hassan et al., "Performance evaluation of raspberry pi as an iot edge signal processing device for a real-time flash flood forecasting system," *International Journal of Advanced Computer Science and Applications*, vol. 13, no. 10, 2022.

[46] Eclipse Foundation, *Eclipse paho MQTT Python client*, https://eclipse.dev/paho/index.php?page=clients/python/index.php, 2013.

[47] InfluxData, *InfluxDB: Time series database*, https://www.influxdata.com/products/influxdb/, 2013.

[48] U. Hunkeler, H. L. Truong, and A. Stanford-Clark, "Mqtt-s—a publish/subscribe protocol for wireless sensor networks," in *2008 3rd International Conference on Communication Systems Software and Middleware and Workshops (COMSWARE'08)*, IEEE, 2008, pp. 791–798.

[49] Grafana Labs, *Grafana: The open observability platform*, https://grafana.com/, 2014.

[50] Z. A. Aziz, D. N. Abdulqader, A. B. Sallow, and H. K. Omer, "Python parallel processing and multiprocessing: A review," *Academic Journal of Nawroz University (AJNU)*, vol. 10, no. 3, 2021.

Appendix A

Appendix

The source code included in this appendix corresponds to the modified version of DINASORE developed in the context of this thesis. The original DINASORE framework was developed by the SYSTEC research group at the University of Porto. The original source code is publicly available at https://github.com/DIGI2-FEUP/dinasore.

The modifications introduced in this thesis extend the original framework to support CFB execution, as described in Chapter 4. This version is present in this repository https://github.com/DIGI2-FEUP/DINASORE4CFB