POLITECNICO DI TORINO

MASTER's Degree in Computer Engineering - Computing and Network Infrastructures

MASTER's Degree Thesis

Orchestration of Monitoring Services for the Electrical Grid Sector Through Multi-Cluster and Cloud Technologies

Supervisors
Prof. FULVIO GIOVANNI OTTAVIO RISSO
Prof. STEFANO GALANTINO

Candidate
DARIO BAGNARA

MARCH 2026

Summary

The increasing complexity of modern electrical power systems requires advanced monitoring infrastructures capable of guaranteeing low latency, scalability, and resilience across geographically distributed environments. In this context, Phasor Data Concentrators (PDCs) play a crucial role in aggregating and forwarding time-synchronized measurements generated by Phasor Measurement Units (PMUs). The placement of PDCs within the communication network directly affects end-to-end latency, bandwidth utilization, and overall system stability.

This thesis addresses the problem of latency-aware PDC placement through a twofold approach that combines graph-based optimization models with cloud-native multi-cluster orchestration in a computing continuum perspective. First, the monitoring network is modeled as a constrained graph, where nodes and links are characterized by latency, bandwidth, and availability attributes. Three placement strategies, Brute Force, Greedy (Dijkstra-based), and Random, are implemented and compared in terms of optimality, computational complexity, stability, and scalability.

Second, the theoretical placement results are integrated into a real Kubernetes-based multi-cluster environment. A fully automated pipeline is developed to transform abstract PMU–PDC–Control Center paths into concrete deployments, including cluster provisioning, database initialization through Percona XtraDB Cluster, and automated configuration of openPDC instances via a custom Command-Line Interface. This integration enables deterministic regeneration of the infrastructure and supports adaptive reconfiguration under dynamic network conditions.

By embedding placement decisions into an intent-based orchestration workflow, the system translates high-level performance objectives, such as latency constraints and connectivity requirements, into automated infrastructure actions. This approach aligns with computing continuum principles, where distributed resources across edge and cloud domains are dynamically orchestrated according to application-level intents.

The experimental evaluation analyzes temporal performance, topological stability, measured through Jaccard distance, and scalability with respect to network size. Results show that while exhaustive approaches guarantee optimality, heuristic

strategies provide a more favorable trade-off between execution time and structural stability in dynamic environments.

Overall, this work demonstrates the feasibility of integrating latency-aware optimization, intent-driven orchestration, and cloud-native technologies to support adaptive placement and coordination of data concentrators in distributed smart grid infrastructures.

Table of Contents

List of Tables VIII

List of Figures IX

Acronyms XI

1 Introduction 1
- 1.1 Context and Motivation . . . 1
- 1.2 Thesis Structure . . . 2

2 State of the Art 7

3 Graph modeling and Algorithms 10
- 3.1 Problem Definition and Constraints . . . 10
- 3.2 Graph Modeling . . . 11
- 3.3 Placement Algorithms . . . 13
  - 3.3.1 Brute Force . . . 13
  - 3.3.2 Greedy approach (Dijkstra-based) . . . 14
  - 3.3.3 Random placement . . . 16
- 3.4 Visualization . . . 18
- 3.5 Test and initial consideration . . . 20
  - 3.5.1 Brute Force . . . 20
  - 3.5.2 Greedy approach . . . 21
  - 3.5.3 Random strategy . . . 23
- 3.6 Algorithm Comparison . . . 24

4 Manual Deployment 28
- 4.1 First system structure and data flow . . . 28
  - 4.1.1 Lower namespaces . . . 29
  - 4.1.2 Higher namespace . . . 29
- 4.2 Logical architecture and data flow . . . 30

4.3 Initial manual deployment using openPDC Manager 31

5 Automation through CLI 32
5.1 Analysis of the openPDC database 32
5.2 Configuration automation via CLI 34
5.2.1 createaccount command 35
5.2.2 createhistorian command 36
5.2.3 addpmu command 36
5.2.4 createoutputstream command 37
5.2.5 connectiontopdc command 38
5.2.6 Command-line configuration workflow 39

6 Integration of placement algorithms within Kubernetes 42
6.1 Automation Pipeline 42
6.1.1 Entry point: autopdc_configurator.py 43
6.1.2 Infrastructure Deployment: deployer.sh 44
6.1.3 PDC Configuration: applier.py 45
6.2 Automatic Regeneration of the Environment 46
6.2.1 Computing Continuum Perspective 46

7 Testing and final evaluation 47
7.1 Objectives of the Experimental Evaluation 47
7.2 Testbed and Experiment Automation 48
7.2.1 Experimental Environment 49
7.2.2 Run Automation 50
7.2.3 Test Scenario 50
7.3 Evaluetion Metrics 52
7.3.1 Temporal Metrics 52
7.3.2 Topological Stability Metrics 53
7.3.3 Scalability Metrics 53
7.4 Experimental Results 54
7.4.1 Experimental Validation of Network Delay Enforcement . . . 55
7.4.2 Temporal Performance under Dynamic Conditions 56
7.4.3 Topological Stability Analysis 59
7.4.4 Scalability Analysis 61
7.5 Discussion and Overall Assessment 66

8 Future Work 67
8.1 Optimization and Stability Improvements 67
8.1.1 Maintenance-Aware PDC Placement 67
8.1.2 Sensitivity Analysis on Latency Constraints 68
8.2 Infrastructure Evolution within the Computing Continuum 69

8.2.1 Real-Time Network Monitoring Integration 69
8.2.2 Hierarchical Multi-Output Extensions in openPDC 69
8.3 Intelligent and Learning-Based Placement Strategies 70
8.3.1 Reinforcement Learning and Graph-Based Approaches . . . 70

9 Conclusions 71

A Automation pipeline 73

Bibliography 77

List of Tables

3.1 Comparison of the three PDC placement algorithms 27

7.1 Comparative Summary of Placement Algorithms 66

List of Figures

3.1 Brute-force strategy 21
3.2 Brute Force with high latency 22
3.3 Greedy strategy 23
3.4 Greedy strategy with down link 24
3.5 Random strategy 25
3.6 Random with splitting enable 26

7.1 Experimental validation of network delay enforcement. The figure shows the network topology used for validation (left) and the corresponding end-to-end latency measured through the OpenPDC Manager (right). 56
7.2 End-to-end execution time decomposition per topology change iteration 57
7.3 End-to-end execution time distribution over 20 runs 58
7.4 Jaccard distance evolution across topology changes. 59
7.5 Jaccard distance distribution over 20 runs 60
7.6 Placement computation time as a function of the number of nodes (logarithmic scale). 62
7.7 Distribution of placement computation time over 20 runs (logarithmic scale). Timeout occurrences are indicated above the corresponding configurations. 63
7.8 Number of selected PDCs (including the CC) as the number of candidate nodes increases. 64
7.9 Distribution of the number of selected PDCs (including the CC) over 20 runs as the number of candidate nodes increases. 65

Acronyms

PMU
: Phasor Measurement Unit

PDC
: Phasor Data Concentrator

AI
: Artificial Intelligence

CC
: Control Center

Chapter 1

Introduction

1.1 Context and Motivation

The growing complexity of modern electrical power systems makes it essential to adopt advanced monitoring infrastructures capable of guaranteeing stability, reliability, and resilience. Among the key components of these infrastructures are Phasor Measurement Units (PMUs), which provide time-synchronized measurements of electrical quantities—such as voltage, current, and phase angle—sampled at high rates. These measurements are precisely aligned in time, enabling operators to observe the real-time dynamic behavior of the electrical grid.

However, the large amount of data generated by a wide deployment of PMUs introduces significant challenges related to data collection, aggregation, and end-to-end transmission latency. To efficiently manage this data flow, Phasor Data Concentrators (PDCs) are employed within the monitoring architecture. Each PDC aggregates data from multiple PMUs, performs preliminary validation and time alignment, and forwards the synchronized data streams either to higher-level concentrators or directly to the Control Center, where advanced analysis and visualization take place.

Since PMU data streams are used by the Control Center for real-time monitoring, protection, and control applications, the timeliness and reliability of data delivery are critical. Excessive latency, packet loss, or loss of time synchronization introduced at the PDC level may result in delayed or inaccurate situational awareness, reducing the operator's ability to promptly detect disturbances or react to abnormal grid conditions. In modern power systems, monitoring infrastructures must operate under non-ideal conditions, including network congestion, partial failures, and dynamically changing topologies. In this context, a suboptimal placement of PDCs may amplify the impact of such events, creating bottlenecks or single points of failure that compromise data consistency and overall system resilience.

The placement of PDCs within the communication network therefore plays a crucial role, as it directly influences not only total latency and bandwidth consumption, but also the robustness and reliability of the monitoring infrastructure. An inefficient configuration may lead to excessive delays, loss of synchronization, or reduced fault tolerance, ultimately affecting the effectiveness of wide-area monitoring and control functions.

This thesis was developed in collaboration with RSE (Ricerca sul Sistema Energetico), an Italian research organization focused on innovation in the energy and smart grid sectors. The goal of the project is to study and design methods to enhance the orchestration and scalability of monitoring services in the electrical grid domain, leveraging modern cloud-native and multi-cluster technologies.

The work is organized into three main phases:

- Definition of a graph-based model to represent the PMU–PDC–Control Center communication network, along with the study of algorithms for optimal PDC placement.
- Deployment of a distributed monitoring architecture based on *k3s* (lightweight Kubernetes) and automatic configuration of PDC instances through a command-line interface (CLI).
- Integration of placement algorithms within the Kubernetes orchestrator to enable adaptive and automated management of PDC nodes across multiple clusters.

By addressing both the modeling and deployment aspects of PDC orchestration, this thesis aims to provide a comprehensive framework that combines optimization, automation, and cloud-native technologies for next-generation smart grid monitoring systems.

1.2 Thesis Structure

This document is structured as follows.

- *Chapter 2* provides the theoretical foundations and a comprehensive review of the state of the art relevant to this work. It introduces the edge–cloud continuum paradigm and discusses the challenges associated with orchestrating latency-sensitive applications across geographically distributed infrastructures. Particular attention is devoted to the limitations of traditional Kubernetes schedulers, which primarily focus on computational resources such as CPU and memory while neglecting real-time network metrics. The chapter then analyzes existing research efforts aimed at extending orchestration frameworks

with network-awareness capabilities, including telemetry-driven scheduling and multi-cluster deployment strategies. Special emphasis is placed on works related to smart grid monitoring systems, highlighting how resilience, scalability, and latency requirements are addressed in current solutions. Through this analysis, a clear research gap is identified: the lack of integrated mechanisms that jointly consider real-time network latency, multi-cluster orchestration, and adaptive placement strategies within a unified framework. By combining theoretical background and critical analysis of existing solutions, Chapter 2 establishes the conceptual context for the latency-aware multi-cluster placement approach proposed in this thesis.

- *Chapter 3* focuses on the graph-based modeling of the monitoring infrastructure and on the algorithms developed for optimal PDC placement. It defines the characteristics of nodes and links, the optimization objectives, such as minimizing latency, reducing bandwidth usage, and limiting the number of concentrators, and the adopted strategies, ranging from simple random placement to heuristic approaches based on Dijkstra's algorithm and exhaustive brute-force exploration.

- *Chapter 4* describes the initial deployment of the monitoring architecture on a Kubernetes-based platform, representing the first concrete implementation of the proposed multi-level PDC infrastructure. The objective of this phase was to design and validate a functional prototype based on openPDC, an open-source implementation of a Phasor Data Concentrator, capable of reproducing the complete data flow from simulated PMUs to a centralized collector. The chapter details the creation of a two-level hierarchical architecture deployed across separate Kubernetes namespaces, each hosting a dedicated PDC instance and its associated database backend. Lightweight k3s clusters are used to provide a containerized and portable orchestration environment, enabling the deployment of openPDC components in a modular and scalable manner. Particular attention is given to networking aspects, service exposure, and port forwarding, which are necessary to enable communication between clusters and external management tools. A central component of the architecture is the Percona XtraDB Cluster (PXC), adopted as the database backend for openPDC. The chapter explains how PXC ensures consistency, replication, and high availability of configuration and measurement data, and describes its integration with Kubernetes through the Percona Operator. The interaction between PDC instances and the database layer is analyzed to provide a comprehensive understanding of the system's internal mechanisms. Initially, the entire configuration process is performed manually through the openPDC Manager graphical interface. This exploratory phase allows a detailed examination of how PMUs and PDCs are logically connected and represented at the

database level. However, the manual approach reveals significant limitations in terms of scalability, reproducibility, and operational reliability, particularly when managing multiple interconnected components. These limitations motivate the transition toward automation. The chapter therefore concludes by outlining the development of a command-line configuration tool capable of programmatically reproducing the manual setup. This automation marks the evolution from a proof-of-concept deployment to a reproducible and scalable pipeline, laying the foundation for the fully automated multi-cluster integration presented in the following chapter.

- *Chapter 5* presents the development of a dedicated Command-Line Interface (`cli_openpdc.sh`) designed to automate the configuration and management of openPDD instances within Kubernetes. It explains how the tool replicates, through structured SQL operations, the actions normally performed via the graphical interface, such as creating user accounts, registering PMUs, defining historians[1], and linking PDCs. This automation transforms the configuration process into a reproducible, scalable, and fully scriptable workflow, paving the way for large-scale and multi-cluster deployments.

- *Chapter 6* introduces the integration of the proposed PDC placement algorithms within a Kubernetes-based orchestration environment. After defining and validating the placement strategies at the modeling level, this chapter addresses the practical challenge of transforming algorithmic decisions into a fully operational multi-cluster infrastructure. In particular, the chapter presents the software architecture that bridges the optimization logic and the orchestration layer, detailing the automated pipeline responsible for infrastructure provisioning, PDC deployment, and hierarchical configuration. The interaction between the placement module and the Kubernetes environment is described in depth, highlighting how abstract placement paths are translated into concrete cluster creation, pod deployment, database initialization, and openPDC configuration steps. Special attention is devoted to the design of a reproducible and deterministic automation workflow, capable of regenerating the entire environment after topology changes. The section explains how deployment and configuration dependencies are resolved through topological ordering and validation mechanisms, ensuring consistency across multiple clusters and preventing configuration conflicts. In this way, the chapter represents the transition from theoretical algorithm design to practical system implementation, demonstrating the feasibility of deploying adaptive PDC placement

[1]In openPDC, historians are components responsible for storing and managing time-series measurement data collected from PMUs for archival, analysis, and retrieval purposes.

strategies in a real Kubernetes-based computing continuum environment.

- *Chapter 7* presents the testing methodology and the final experimental evaluation of the proposed PDC placement strategies. The chapter assesses the behavior of the considered algorithms in a realistic Kubernetes-based orchestration environment, with the objective of evaluating not only their computational properties, but also their operational impact in dynamic deployment scenarios. In particular, the chapter describes the experimental testbed, the automation framework used to execute repeated and reproducible evaluation campaigns, and the generated test scenarios adopted to analyze both dynamic topology changes and scalability conditions. It also introduces the evaluation metrics used to measure temporal performance, topological stability, and scalability. The chapter then reports the experimental results, starting with the validation of the network delay enforcement mechanism and proceeding with a comparative analysis of the placement algorithms under dynamic conditions and increasing topology size. The results highlight the trade-offs among optimality, reaction time, structural stability, and scalability, showing the practical strengths and limitations of Brute Force, Greedy, and Random placement strategies in real deployment environments. Overall, the chapter provides a systematic experimental evaluation of the proposed framework, clarifying the strengths and limitations of the considered placement strategies under realistic deployment conditions.

- *Chapter 8* presents the main directions for future research emerging from this work. Building upon the implemented placement framework, the chapter discusses possible extensions aimed at enhancing adaptability, scalability, and operational robustness in computing continuum environments. In particular, it explores the integration of real-time network monitoring mechanisms into the orchestration loop, enabling placement decisions to be triggered by live telemetry data rather than predefined topology changes. It also analyzes the introduction of maintenance-aware optimization criteria, where placement strategies explicitly minimize structural modifications across successive iterations in order to reduce operational churn. Additional research perspectives include sensitivity analysis with respect to latency constraints, architectural extensions of openPDC to support controlled multi-output configurations, and the adoption of learning-based approaches for adaptive placement. Through these considerations, Chapter 8 outlines how the proposed framework can evolve toward more autonomous, intent-driven, and self-adaptive orchestration models.

- *Chapter 9* concludes the thesis by synthesizing the overall contributions and reflecting on the broader implications of the proposed approach. It highlights

how the integration of graph-based optimization, automated configuration pipelines, and Kubernetes-native orchestration enables the deployment of latency-sensitive monitoring systems across distributed infrastructures. The chapter emphasizes the conceptual advancement represented by the fusion of placement algorithms and intent-based orchestration mechanisms, framing the proposed solution within the broader vision of the computing continuum. Finally, it discusses the relevance of this work for smart grid monitoring applications and outlines how the presented framework contributes toward resilient, adaptive, and network-aware multi-cluster infrastructures.

In summary, this work aims to combine theoretical modeling and practical deployment to provide a complete and scalable framework for the orchestration of monitoring services in the electrical grid domain.

Chapter 2

State of the Art

In recent years, edge and fog computing paradigms have gained increasing relevance in addressing stringent requirements related to geographical proximity, end-to-end latency reduction, and enhanced data privacy. As highlighted by several studies [1][2], these paradigms are no longer confined to peripheral devices, but are progressively extending their scope to include small and medium-scale data centers located at the network edge. This evolution enables the reuse of shared infrastructural components and improves service adaptability, while also supporting a smoother transition between edge and cloud environments [3], further fostered by the growing adoption of multi-region cloud infrastructures.

Alongside this evolution, a substantial body of research has focused on the development of orchestration and scheduling mechanisms aimed at optimizing computational resources such as CPU and memory [4]. However, the heterogeneous and dynamic nature of the edge–cloud continuum introduces challenges that traditional orchestrators struggle to address. In particular, most existing solutions lack native mechanisms for dynamically adapting to changing network conditions and do not incorporate real-time network metrics into their decision-making processes. In edge–cloud scenarios, these limitations become particularly critical, as scheduling decisions can have a substantial impact on application performance, especially for latency-sensitive workloads managed through Kubernetes.

Recent works have attempted to mitigate these limitations by extending existing orchestration frameworks with network-awareness capabilities. Specifically, the solutions presented in [5] and [6] introduce custom Kubernetes plugins designed to support telemetry-aware schedulers and network-aware pod placement strategies. Other approaches, such as the one proposed in [7], focus on pod offloading within multi-tier architectures that combine cloud and edge resources in order to minimize overall application latency. While these solutions demonstrate the benefits of considering network conditions during scheduling, they often rely on static assumptions or limited network models and do not fully address dynamic,

multi-cluster scenarios.

In this context, the design of cost-aware orchestration algorithms plays a fundamental role in reducing deployment overhead and limiting the consumption of external resources by applications. In distributed infrastructures, such overhead should not be treated independently of latency, but instead modeled as network overhead, since geographically distant resources increase both communication delays and data transfer and processing requirements. Although cost minimization has been extensively studied in traditional cloud computing environments, these approaches typically assume computational nodes that are physically co-located.

In contrast, truly distributed edge–cloud scenarios involve computational resources spread over wide geographical areas, where microservice-based applications can be dynamically partitioned and deployed either at the edge or in the cloud depending on the use case. For instance, studies such as [8] and [9] assume that applications are deployed within geographically constrained regions, which limits their applicability in large-scale, geographically distributed environments. Furthermore, several existing works on resource allocation for application deployment [10][11] do not jointly address microservice placement and their communication requirements. As a result, such solutions are only partially effective in highly distributed scenarios, where application deployment imposes fine-grained spatial constraints that translate into specific latency and bandwidth requirements to ensure the expected quality of service.

Beyond general-purpose edge–cloud orchestration, several thesis works have specifically addressed the challenges of deploying resilient and low-latency services in smart grid environments by leveraging edge and fog computing paradigms. In particular, the work presented in [12] investigates the deployment of virtualized services close to data sources in order to support real-time control loops, emphasizing the importance of resilience against hardware failures, software faults, and network partitioning. The study highlights the potential of Kubernetes-based multi-cluster solutions to orchestrate services across thousands of geographically distributed sites, but primarily focuses on architectural resilience rather than on fine-grained, latency-driven placement decisions.

Similarly, the thesis in [13] explores an Edge-to-Cloud multi-cluster orchestration model for smart grid monitoring services, leveraging Kubernetes and the Liqo framework to enable dynamic workload relocation and island-mode operation. The proposed architecture demonstrates strong resilience properties and scalability, showing that distributed cluster federation can effectively support smart grid applications without significant latency degradation. However, scheduling decisions are largely driven by architectural and fault-tolerance considerations, while network latency is not explicitly integrated as a real-time optimization metric within the orchestration loop.

A complementary perspective is provided in [14], which focuses on data-centric

communication models for smart grid environments, analyzing publish–subscribe paradigms to ensure data availability, reliability, and bounded latency. While this work provides valuable insights into communication architectures and Quality of Service management, it does not directly address the problem of dynamic application placement across distributed compute infrastructures.

Taken together, these works clearly demonstrate the relevance of edge–cloud architectures and multi-cluster orchestration for smart grid applications, as well as the central role of latency and resilience in such environments. Nevertheless, most existing approaches either rely on static or coarse-grained network models, or assume centralized control planes that do not scale well across geographically distributed clusters. Moreover, the integration of real-time network latency measurements into multi-cluster scheduling decisions remains largely unexplored, especially in scenarios where applications span multiple administrative domains and heterogeneous infrastructures.

Empirical evidence supporting the benefits of network-aware scheduling in distributed Kubernetes environments is provided in [15]. The authors evaluate custom scheduling plugins on Kubernetes clusters spanning multiple network zones, showing that the integration of real-time latency measurements into the scheduling process consistently reduces execution time for communication-intensive workloads, including analytics and machine learning tasks. Their results confirm that network-aware scheduling improves not only performance but also predictability, particularly in scenarios characterized by high inter-service communication. Further work in this direction is presented in [16], where a latency-aware scheduler for Kubernetes is developed and evaluated in a geographically distributed multi-cluster environment. Unlike default Kubernetes scheduling, which primarily considers CPU and memory constraints, this approach incorporates end-to-end latency measurements into placement decisions to enhance Quality of Experience for latency-sensitive applications.

From the analysis of the state of the art, a clear research gap emerges. While several approaches address resource orchestration, latency optimization, and resilience independently, few solutions jointly consider real-time network latency, multi-cluster deployments, and closed-loop adaptation mechanisms within a unified scheduling framework. This gap is particularly evident for data-intensive and latency-sensitive applications operating across geographically distributed infrastructures, such as those based on phasor measurement units.

Building upon these existing works, this thesis addresses the identified limitations by proposing a latency-aware, multi-cluster scheduling framework that operates as a closed-loop control system. By continuously monitoring network conditions and dynamically adapting application placement decisions, the proposed solution aims to extend current approaches toward practical deployment in large-scale edge–cloud infrastructures.

Chapter 3

Graph modeling and Algorithms

This chapter introduces the modeling and algorithmic foundations adopted for the PDC placement problem.

The monitoring infrastructure is first represented through a graph-based model that captures connectivity, latency, bandwidth, and processing constraints. Based on this representation, several placement algorithms are presented and analyzed, highlighting different strategies for determining feasible PMU–to–Control Center paths and selecting the nodes that host the PDCs.

3.1 Problem Definition and Constraints

The goal of this first phase of the thesis is the modeling of the PDC placement problem in a general and flexible manner. The placement of a Phasor Data Concentrator (PDC) can be formalized as the problem of selecting an optimal set of nodes within a graph representing the network.

In this context, a placement is considered optimal if it satisfies a set of performance and resource constraints while preserving full connectivity between all PMUs and the Control Center (CC). In particular, the solution must comply with a maximum end-to-end *latency constraint*, defined as the sum of link transmission delays and processing times introduced by intermediate PDC nodes along each PMU–to–CC path.

Additionally, the placement must ensure that *bandwidth constraints* are respected on every communication link. The aggregated traffic generated by multiple PMUs must not exceed the available capacity of any edge in the graph, thereby preventing congestion and guaranteeing stable data transmission.

The optimization objective therefore consists in identifying a configuration that

minimizes overall latency and resource utilization while satisfying connectivity, latency, and bandwidth constraints. This trade-off between delay minimization and efficient bandwidth usage defines the core of the optimization problem addressed in this work and reflects the practical requirements of scalable and reliable monitoring infrastructures.

Due to the combinatorial nature of node selection and path assignment, the problem becomes increasingly complex as the size of the network grows, motivating the need for both exact and heuristic placement strategies.

3.2 Graph Modeling

The approach followed during the first stage of the project was to start with a set of Python scripts aimed at defining a *"base" graph model*. The objective was to create a topology that could represent a realistic network while remaining simple enough for testing and debugging. For this reason, a small but highly connected graph was chosen, not fully meshed, since that configuration would be too restrictive and unrealistic. A full-mesh topology would allow every node to communicate directly with all others, but even a single fault would compromise the behavior of most algorithms. Instead, a partially connected graph provides a more balanced and representative test environment, offering redundancy without unnecessary complexity.

The choice of a graph-based representation is particularly suitable because it provides a clear and abstract view of the monitoring network. Each node in the graph represents a *computational node*, that is a location in which a component such as a PMU, a PDC, or the Control Center can be deployed. Intermediate network elements such as routers or switches are not explicitly modeled since their internal behavior does not affect placement decisions. What matters is the logical connectivity between nodes: if two nodes A and B are connected by an edge, it means that they can communicate directly, or in practical terms, that a network connection between them is available and operational (for example, a successful ping). This abstraction makes the model both flexible and scalable while preserving the essential communication relationships that influence PDC placement.

In this model, therefore, the path between a PMU and the Control Center must always form a chain of PDCs. Each edge is treated as a *logical connection* rather than a purely physical one, as it may represent the set of intermediate network components that enable communication between the two endpoints. This simplification allows the focus to remain on the key aspects of optimization, such as latency, bandwidth, and processing, without being limited by low-level network details.

Each node in the graph is characterized by several attributes that describe its

computational and operational features:

- *group*: the group or cluster to which the node belongs
- *level*: the hierarchical level of the node within the network
- *processing*: processing capability or computational power
- *memory*: available RAM (in GB)
- *storage*: disk capacity (in GB)
- *status*: operational state (online, offline, maintenance)
- *energy*: energy consumption (in kWh)

Similarly, each edge is defined by the following parameters:

- *latency*: the transmission delay of the connection (in ms)
- *bandwidth*: the available bandwidth (in Mbps)
- *status*: connection state (up, down)
- *type*: the link technology (fiber, ethernet, wireless)

These attributes are essential for simulating a realistic environment in which both computational capacity and communication constraints influence the final placement of the PDCs. By adjusting these parameters, it is possible to reproduce different operating conditions and evaluate how the placement strategies behave under various network configurations. Finally, a fundamental structural constraint is introduced in the graph model. Due to the intrinsic behavior of Phasor Data Concentrators, when two or more data paths originating from different PMUs converge at the same PDC, they must continue along a common path towards the Control Center. In other words, once multiple flows are aggregated at a given PDC, they cannot be split again across different downstream paths. This constraint explicitly prevents *path splitting* after convergence and enforces a tree-like structure rooted at the Control Center, reflecting the actual data aggregation process performed by PDCs.

Overall, this graph model provides a flexible yet realistic abstraction of the monitoring infrastructure, capturing the essential computational, communication, and structural constraints that govern PDC placement, and serving as a solid foundation for the optimization algorithms discussed in the following sections.

3.3 Placement Algorithms

Once the base graph and the main constraints were defined, the next step was the selection and implementation of several predefined *placement algorithms* designed to identify the most suitable nodes for PDC deployment. Each algorithm follows a different strategy to explore feasible PMU–to–Control Center paths and to allocate PDCs under latency, bandwidth, and availability constraints.

The following subsections describe the considered placement strategies in detail, highlighting their underlying principles, computational characteristics, and expected trade-offs.

3.3.1 Brute Force

The brute force approach performs an *exhaustive exploration* of the solution space by systematically evaluating all possible PDC placement configurations. Starting from a single PDC and progressively increasing the number of deployed concentrators, the algorithm enumerates all subsets of candidate nodes and, for each configuration, evaluates every feasible combination of PMU–to–Control Center paths. Only paths that form valid PMU–PDC–CC chains, satisfy the maximum end-to-end latency constraint, respect bandwidth limitations, and comply with node and link availability are considered. When the no-splitting constraint is enforced, the algorithm additionally ensures that all paths converging at the same PDC share an identical downstream suffix toward the Control Center.

For each candidate configuration, the algorithm selects the combination of paths that maximizes PMU coverage while minimizing the total aggregated latency.

Although this exhaustive evaluation guarantees the identification of an optimal solution within the modeled constraints, its combinatorial complexity grows rapidly with the number of candidate nodes and PMUs, making the approach impractical for large or highly connected graphs.

For this reason, the Brute Force algorithm is adopted as an optimal reference solution and is primarily used to assess the quality of the more scalable placement strategies introduced in the following subsections.

For completeness, the pseudocode of the Brute Force algorithm is reported below, providing a concise and structured representation of its logical workflow.

Algorithm: Brute Force PDC Placement

Require: Graph G, maximum latency $L_{\max}$
Ensure: Best PDC set $\mathcal{P}^*$ and PMU paths Π^*
1: $\mathcal{M} \leftarrow$ set of PMU nodes
2: $\mathcal{C} \leftarrow$ set of candidate nodes
3: $cc \leftarrow$ control center node

$\mathcal{P}^* \leftarrow \emptyset$, $\Pi^* \leftarrow \emptyset$
$BestCovered \leftarrow -1$, $BestLatency \leftarrow \infty$, $BestK \leftarrow \infty$
for $k = 1$ to $|\mathcal{C}|$ **do**
    for all subsets $\mathcal{P} \subseteq \mathcal{C}$ with $|\mathcal{P}| = k$ **do**
        $FeasiblePaths \leftarrow \emptyset$
        $CoveredPMUs \leftarrow \emptyset$
        for all $m \in \mathcal{M}$ **do**
            $Allowed \leftarrow \mathcal{P} \cup \{m, cc\}$
            $G' \leftarrow$ subgraph of G induced by $Allowed$
            $Candidates \leftarrow$ simple paths $m \rightarrow cc$ in G'
            $ValidPaths \leftarrow \emptyset$
            for all $p \in Candidates$ **do**
                compute delay $d(p)$ (edge latency + PDC processing)
                if p respects chain structure and $d(p) \leq L_{\max}$ **then**
                    add p to $ValidPaths$
                end if
            end for
            if $ValidPaths \neq \emptyset$ **then**
                store $ValidPaths$ for m
                add m to $CoveredPMUs$
            end if
        end for
        if $CoveredPMUs = \emptyset$ **then**
            continue
        end if
        for all combinations selecting one path per $m \in CoveredPMUs$ **do**
            if bandwidth and splitting constraints are satisfied **then**
                compute total latency
                $CoveredCount \leftarrow |CoveredPMUs|$
                if $CoveredCount > BestCovered$ **then**
                    update best solution
                else if $CoveredCount = BestCovered$ and latency improves **then**
                    update best solution
                else if $CoveredCount = BestCovered$ and latency equal and $k < BestK$ **then**
                    update best solution
                end if
            end if
        end for
    end for
end for
return $\mathcal{P}^*, \Pi^*$

3.3.2 Greedy approach (Dijkstra-based)

This algorithm looks for an *optimal solution locally*. Starting from the active subgraph, obtained by filtering out offline nodes and unavailable links, the algorithm

processes each PMU independently and selects the first feasible path to the Control Center (CC) among the shortest simple paths ordered by increasing latency. A path is considered feasible if it satisfies the maximum end-to-end latency constraint, the bandwidth constraints on all traversed links, and an optional no-splitting constraint, which enforces a unique downstream direction for each candidate PDC.

Once a valid path is found, the algorithm immediately commits the corresponding resource allocation and moves to the next PMU, without revisiting previous decisions. This greedy, no-backtracking behavior ensures low computational complexity but may lead to suboptimal solutions due to local optima.

To partially mitigate this limitation, an alternative version of the algorithm with backtracking has also been implemented. In this variant, previously selected paths may be reconsidered when no feasible solution can be found for subsequent PMUs, improving solution quality at the cost of increased computational complexity. The impact of introducing backtracking is analyzed in the experimental evaluation, highlighting how it affects the resulting PDC placement and identifying the scenarios in which reconsidering previous decisions leads to significantly improved solutions.

The logical structure of the greedy placement algorithm is reported below, highlighting the sequence of operations that govern the placement process.

Algorithm: Greedy PDC Placement with Limited Backtracking
Require: Graph G, latency bound $L_{\max}$, split flag s, top-K paths, backtrack depth D
Ensure: Selected PDC set $\mathcal{P}$, PMU paths Π

$H \leftarrow$ ActiveGraph(G)
$CC \leftarrow$ control center in H
$\mathcal{M} \leftarrow$ PMUs in H, with demand $rate(m)$
for all $m \in \mathcal{M}$ **do**
    $\mathcal{C}[m] \leftarrow$ FeasiblePaths($H, m, CC, K, L_{\max}$)
end for
Sort $\mathcal{M}$ by increasing $|\mathcal{C}[m]|$
Init $used\_bw[e] \leftarrow 0$, $\Pi \leftarrow \emptyset$, $\mathcal{P} \leftarrow \emptyset$
if $s = false$ **then**
    Init $next[c] \leftarrow$ undefined for all candidate nodes c
end if
if $s = true$ **then**
    for all $m \in \mathcal{M}$ **do**
        for all $p \in \mathcal{C}[m]$ **do**
            if BWFeasible($p, rate(m), used\_bw$) **then**
                AddBW($p, rate(m), used\_bw$); $\Pi[m] \leftarrow p$
                $\mathcal{P} \leftarrow \mathcal{P} \cup Cand(p)$; **break**
            end if
        end for
    end for
else
    $i \leftarrow 1$, $stack \leftarrow \emptyset$, $bt\_count \leftarrow 0$

```
    while i ≤ |M| do
        m ← M[i], placed ← false, start ← 1
        if stack ≠ ∅ and Top(stack) refers to m then
            Pop (i, m, j_last) from stack
            start ← j_last + 1
        end if
        for j = start to |C[m]| do
            p ← C[m][j]
            if NoSplitOK(p, next) and BWFeasible(p, rate(m), used_bw) then
                AddBW(p, rate(m), used_bw); SetNext(p, next)
                Π[m] ← p; Push (i, m, j) into stack
                P ← P ∪ Cand(p)
                placed ← true; break
            end if
        end for
        if placed then
            bt_count ← 0; i ← i + 1
        else if stack ≠ ∅ then
            Pop (i', m', j') from stack
            Remove Π[m']; RebuildConstraints(Π, used_bw, next)
            Push (i', m', j') back into stack
            bt_count ← bt_count + 1
            if bt_count > D then
                bt_count ← 0; i ← i + 1
            else
                i ← i'                                  ▷ retry m' from path j' + 1
            end if
        else
            i ← i + 1
        end if
    end while
end if
return P, Π
```

3.3.3 Random placement

The random placement algorithm explores feasible PMU–to–Control Center paths by performing a *randomized depth-first traversal* of the active graph. For each PMU, multiple candidate paths are generated by randomly selecting adjacent nodes at each step, while enforcing feasibility constraints such as node and link availability, bandwidth limits, and maximum end-to-end latency.

When the splitting constraint is disabled, the algorithm enforces a non-splitting behavior by ensuring that paths converging at the same candidate PDC share a common downstream suffix toward the Control Center, thus preserving the aggregation semantics of PDCs. Conversely, when splitting is allowed, converging

paths are permitted to diverge again downstream, increasing routing flexibility at the cost of potentially higher resource consumption.

Although based on randomized exploration, the strategy is implemented as a *pseudo-random* approach. After the first execution, where paths are selected randomly, the same random seed is reused in subsequent runs. As a result, for a given graph and identical constraints, the algorithm produces consistent placement decisions. Different outcomes occur only when latency or bandwidth constraints invalidate previously selected paths.

In addition, the exploration process is explicitly bounded: the number of sampling attempts and candidate paths evaluated for each PMU is limited by predefined thresholds. This constraint prevents uncontrolled expansion of the search space as the topology grows, ensures predictable computational cost, and limits excessive stochastic variability.

This design avoids inconsistent deployments that could otherwise lead to the creation of unnecessary or logically incoherent cluster configurations.

Among the feasible paths discovered for each PMU, one path is selected and its associated resource usage is immediately committed before processing the next PMU.

This approach enables the exploration of diverse routing configurations with limited computational effort. However, due to the absence of global optimization, it may still result in suboptimal placements. For this reason, the random placement algorithm is used as a comparative baseline in the experimental evaluation, providing a reference point to contextualize the performance of more structured placement strategies.

Below, the logical formulation of the algorithm is provided, offering a structured view of the decision process underlying the random approach.

Algorithm: Random PDC Placement (pseudo-random DFS sampling)
Require: Graph G, latency bound $L_{\max}$, splitting flag s, max tries T, max samples S
Ensure: Selected PDC set $\mathcal{P}$, PMU paths Π, maximum delay $D_{\max}$

$\mathcal{M} \leftarrow$ nodes with role = PMU
$CC \leftarrow$ node with role = CC
Initialize edge usage $BW[e] \leftarrow 0$ for all edges e
Initialize $\Pi \leftarrow \emptyset$, $\mathcal{P} \leftarrow \emptyset$, $Delays \leftarrow \emptyset$
$NoSplit \leftarrow (s = false)$
$Forced \leftarrow \emptyset$ ▷ candidate → forced successor
for all $m \in \mathcal{M}$ **do**
    $rate \leftarrow data\_rate(m)$
    $Cand \leftarrow \emptyset$
    for $t = 1$ to T **do**
        $p \leftarrow \text{RandomDFS}(G, m, CC, BW, rate, NoSplit, Forced)$
        if p undefined **then**
            continue

```
        end if
        d ← Delay(p)
        if d > L_max then
            continue
        end if
        if NoSplit and UpdateForcedSuffix(p, Forced) fails then
            continue
        end if
        Add (p, d) to Cand
        if |Cand| = S then
            break
        end if
    end for
    if Cand empty then
        continue
    end if
    Select (p*, d*) uniformly at random from Cand
    CommitBandwidth(p*, rate, BW)
    Π[m] ← (p*, d*)
    P ← P ∪ Candidates(p*)
    Add d* to Delays
end for
D_max ← max(Delays) or 0
return P, Π, D_max
```

3.4 Visualization

Finally, to support the interpretation and validation of the results obtained, a dedicated visualization function is used to render the graph with the selected PDC placements and communication paths. This function, implemented using the NetworkX and matplotlib.pyplot libraries, provides a comprehensive and expressive graphical representation of the network topology, enriched with semantic and performance-related information.

The function takes as input the graph structure, the set of nodes selected as PDCs, and, when available, the PMU → CC paths computed by the placement algorithm. In order to ensure readability and structural clarity, the nodes are automatically arranged using a hierarchical layout. When the pydot backend is available, a DOT-based layout is used, which emphasizes the directional and layered nature of the PMU → PDC → CC communication chain. In environments where pydot is not available or fails, the function gracefully falls back to a force-directed spring layout, ensuring robustness and portability of the visualization process.

Each node in the graph is visually encoded according to its *role in the system*. The central control node (CC) is highlighted in red to clearly identify the final

aggregation point of all PMU data streams. PMUs are shown in light green, representing data sources at the edge of the network. Nodes selected as PDCs by the algorithm are rendered in orange, making the placement decisions immediately visible, while all remaining candidate or intermediate nodes are colored in light blue. In addition to color coding, node labels are enriched with *contextual information*: PDC nodes include their processing latency, while CC and PMU nodes are explicitly annotated with their role. This combination of visual and textual cues allows the reader to quickly associate each node with both its functional role and its contribution to end-to-end latency.

Edges are similarly annotated to convey quantitative information. Each link is labeled with its associated communication latency, and all edges are initially drawn in a neutral light gray to represent the underlying physical or logical network topology. When PMU $\rightarrow$ CC paths are provided, the function overlays these routes on top of the base graph using distinct colors for each PMU. This highlighting mechanism makes it possible to visually trace the exact sequence of nodes and links used by each PMU to reach the CC, and to immediately observe shared segments where multiple data streams converge on the same PDC chain. Such overlaps are particularly important for validating aggregation behavior and ensuring that routing constraints are correctly enforced.

To further enhance interpretability, the visualization includes a textual *summary panel* embedded directly within the figure. This panel reports, for each PMU, the total end-to-end latency of the selected path toward the CC and indicates whether it satisfies the specified maximum latency constraint. PMUs for which no valid path is available are also explicitly reported. This summary provides a compact yet effective bridge between the graphical representation and the quantitative evaluation metrics used during optimization.

Finally, a legend is added to clearly explain the color and role encoding used in the graph, and the resulting figure is saved to disk at high resolution to ensure suitability for inclusion in reports, presentations, or thesis documents. When an interactive backend is available, the figure can also be displayed on screen, facilitating rapid inspection during development and experimentation.

Overall, this visualization function plays a crucial role beyond mere presentation. It offers an immediate qualitative understanding of the algorithm's behavior, enables visual debugging of placement and routing decisions, and serves as an effective validation tool to confirm that connectivity, bandwidth aggregation, and latency constraints are jointly respected by the proposed solution.

3.5 Test and initial consideration

The algorithms are compared not only against each other, but also in real time with themselves by dynamically changing parameters such as node status, link latency, and available bandwidth. This approach makes it possible to observe how each placement strategy adapts to *different network conditions* and how sensitive it is to variations in the underlying topology.

It is important to emphasize that all tests are conducted on relatively small-scale graphs, executed locally, in order to better analyze the internal behavior of each algorithm and to clearly visualize how decisions are made. At this stage, the experiments are entirely software-based, and Kubernetes is not yet involved in the deployment or orchestration process. Kubernetes will play a central role in the third phase of the thesis (see Chapter 4), where the focus will shift from algorithmic evaluation to real distributed execution and inter-cluster coordination.

The following sections present and discuss the results obtained from these preliminary tests, highlighting the main differences in performance, scalability, and robustness among the implemented placement strategies.

3.5.1 Brute Force

The brute-force approach proved to be highly reliable, as it consistently identifies the optimal PDC placement by exhaustively evaluating all possible deployment configurations. For each test case, the algorithm systematically explores *every combination of candidate nodes* and verifies feasibility with respect to connectivity, node and link availability, bandwidth constraints, and end-to-end latency.

This exhaustive exploration guarantees that the best possible configuration—namely, the one minimizing the total delay between all PMUs and the central collector (CC)—is always selected when a feasible solution exists. However, this optimality comes at a substantial *computational cost*. Since the algorithm evaluates both feasible and infeasible configurations, its complexity grows exponentially with the number of candidate nodes and the degree of network connectivity.

As a consequence, the brute-force approach introduces a significant computational overhead, making it impractical for medium- or large-scale graphs, where the number of potential PMU–to–CC paths increases rapidly. For this reason, it is best suited for small and sparsely connected networks, where the exhaustive search can be completed within acceptable time limits.

Figure 3.1 illustrates the optimal paths selected for each PMU under nominal network conditions. In a second execution, shown in Figure 3.2, an additional delay is introduced on the link between node N5 and the CC. Given the maximum end-to-end latency constraint of 35 ms, the algorithm is no longer able to find a feasible path for PMU3, which therefore remains disconnected from the CC.

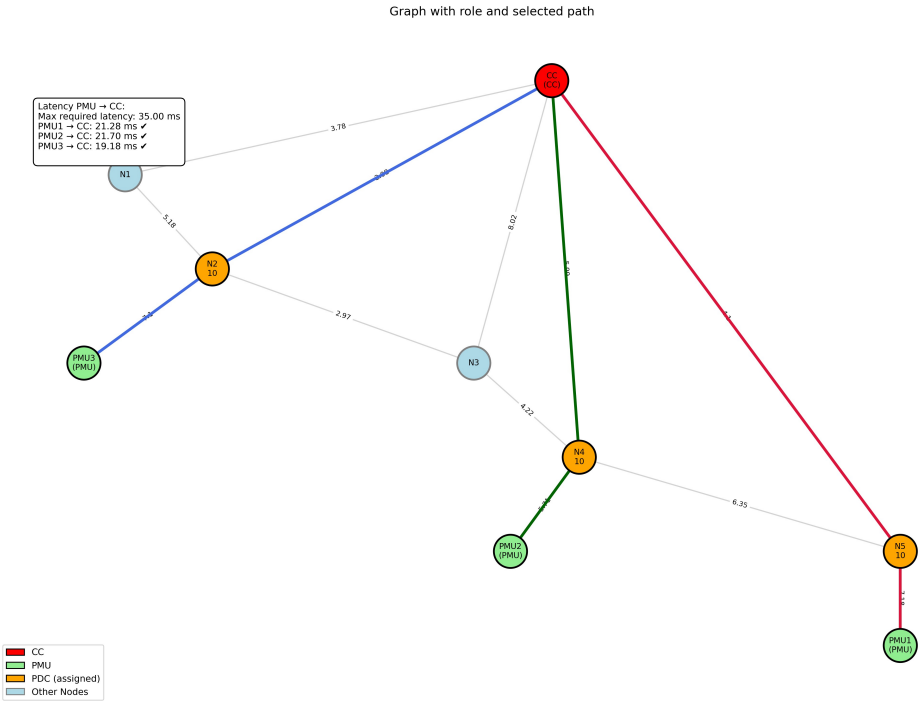

Figure 3.1: Brute-force strategy

3.5.2 Greedy approach

The greedy approach highlights how even minor variations in network conditions, such as an increase in link latency or a change in the operational status of a node or edge, can significantly impact the resulting PDC placement. In particular, the removal or degradation of a single connection often forces the algorithm to deploy a larger number of PDCs in order to preserve connectivity between PMUs and the central collector.

Since the greedy algorithm bases its decisions exclusively on link latency, without explicitly accounting for theprocessing delay introduced by each PDC, it tends to select paths that appear optimal from a transmission-delay perspective only. As a result, when a low-latency link becomes unavailable or its delay increases, the algorithm may switch to alternative routes containing additional PDCs, ultimately leading to a higher overall end-to-end latency once processing delays are considered.

This behavior reflects the *locally optimal* but globally myopic nature of the greedy strategy. While the algorithm reacts quickly to network changes, it lacks awareness of cumulative effects, such as the latency introduced by intermediate processing nodes.

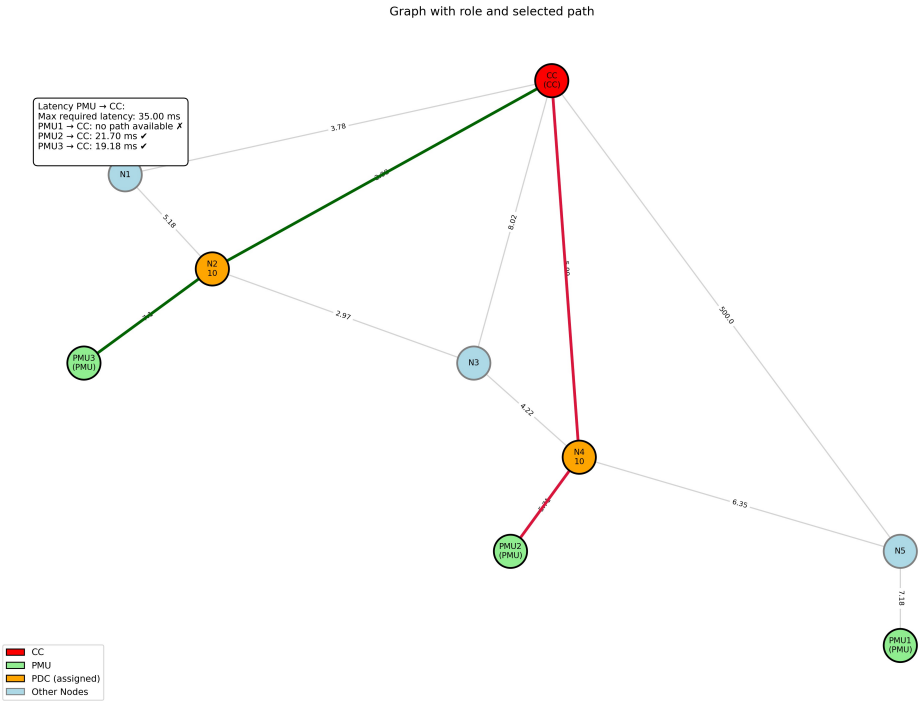

Figure 3.2: Brute Force with high latency

Figure 3.3 shows the result of applying the greedy approach to the original network graph, where all PMU–to–CC paths satisfy the maximum latency constraint and PDCs are deployed on nodes N1, N2, N4, and the CC. Subsequently, the link between N1 and the CC is set to down, and the bandwidth of the N2–CC link is limited to support the traffic of only one PMU. As illustrated in Figure 3.4, these changes force the algorithm to reroute traffic through an alternative path passing via node N3. It is important to note that this behavior is enabled by the use of the greedy variant with backtracking. Without backtracking, since PMU3 is processed after PMU2, the algorithm would fail to find a valid path for PMU3. The direct connection to the CC is unavailable, the N2–CC link can accommodate only one PMU, and traffic splitting at N2 is not permitted. This scenario clearly illustrates the limitations of a purely greedy approach and highlights the importance of reconsidering previous decisions in dynamic network conditions.

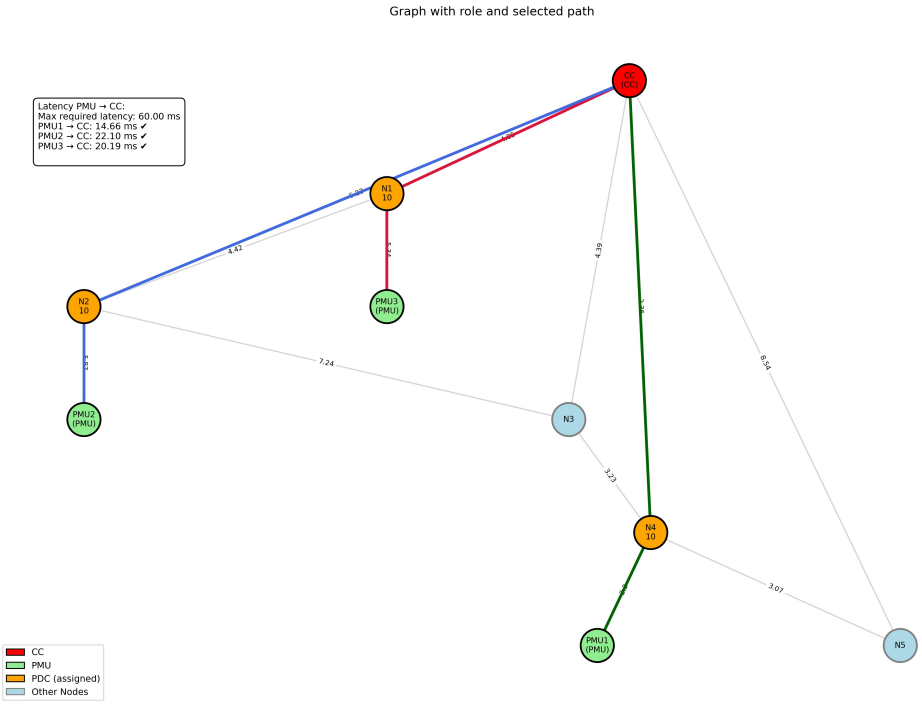

Figure 3.3: Greedy strategy

3.5.3 Random strategy

The results obtained with the random strategy show a clear tendency to deploy an *excessive number of PDCs*, often exceeding what is strictly required to maintain connectivity between PMUs and the central collector. In addition, the selected paths frequently converge toward common PDCs, which are shared by multiple PMU routes.

While this convergence can simplify connectivity, it also leads to inefficient resource utilization and increases the risk of bandwidth saturation on shared links, as multiple data streams are aggregated on the same intermediate nodes. In this context, the role of the *splitting* constraint becomes particularly evident.

To better highlight the limitations of the random approach, a more connected and complex graph was considered, in contrast to the simpler topologies used for the brute-force and greedy strategies. Figure 3.5 shows the result obtained with splitting disabled. In this case, each PDC is constrained to have a single outgoing edge, which limits the growth of the solution: once a PDC is selected, subsequent paths encountering it are forced to follow the same downstream route.

Conversely, when splitting is enabled, as shown in Figure 3.6, paths are allowed

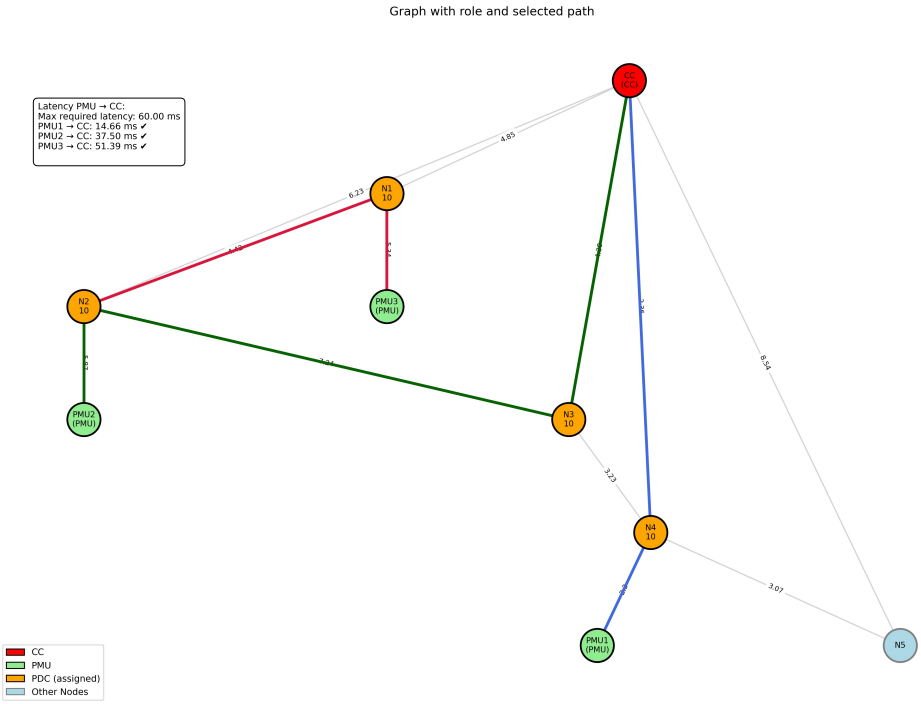

Figure 3.4: Greedy strategy with down link

to diverge after converging at a shared PDC. This increased flexibility leads to a rapid growth in the number of deployed PDCs, clearly exposing the absence of any global optimization objective in the random strategy. Overall, while the random approach serves as a useful baseline for exploring the solution space, its lack of optimization criteria and its sensitivity to path convergence make it less efficient and more resource-intensive compared to the other placement strategies.

3.6 Algorithm Comparison

When comparing the three placement strategies—Brute Force, Greedy, and Random—it becomes evident that each of them addresses a different point in the trade-off space between optimality, computational efficiency, and scalability.

The brute-force approach serves as a reference benchmark for optimality. By exhaustively evaluating all possible PDC placement configurations, it guarantees the identification of the best achievable solution in terms of end-to-end latency and connectivity, whenever a feasible configuration exists. This makes it particularly suitable for small-scale experiments and for validating the correctness and quality

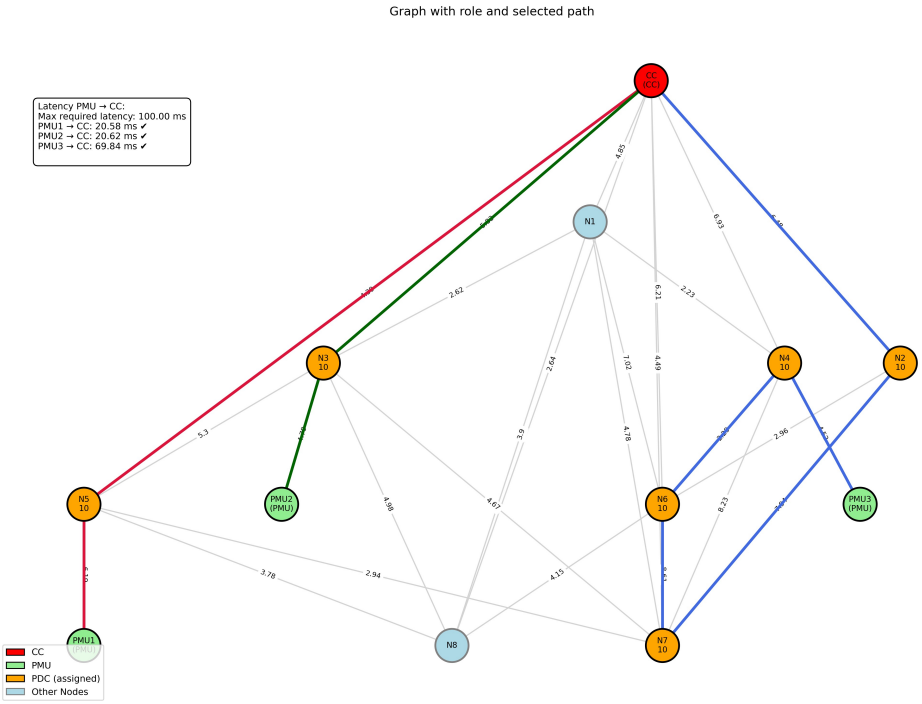

Figure 3.5: Random strategy

of the solutions produced by more scalable algorithms. However, its computational complexity grows exponentially with the number of candidate nodes and PMUs, rendering it impractical for medium- or large-scale networks. As a result, while brute force defines an upper bound in terms of solution quality, it also represents a lower bound in terms of scalability and computational efficiency.

The Greedy approach represents a compromise between optimality and efficiency. By selecting locally optimal paths based on link latency, it achieves very low execution times and is therefore well suited for iterative or near real-time placement scenarios. Its simplicity and reactivity make it attractive in dynamic environments, where rapid adaptation to changing network conditions is required. Nevertheless, due to its local decision-making process and its limited awareness of global effects, such as cumulative processing delays and interactions between multiple PMU paths, the greedy strategy often produces suboptimal global configurations. In particular, network changes may lead to the deployment of additional PDCs or to longer end-to-end paths, affecting both stability and overall latency.

The Random approach lies at the opposite end of the spectrum with respect to structured optimization. By exploring the solution space without deterministic

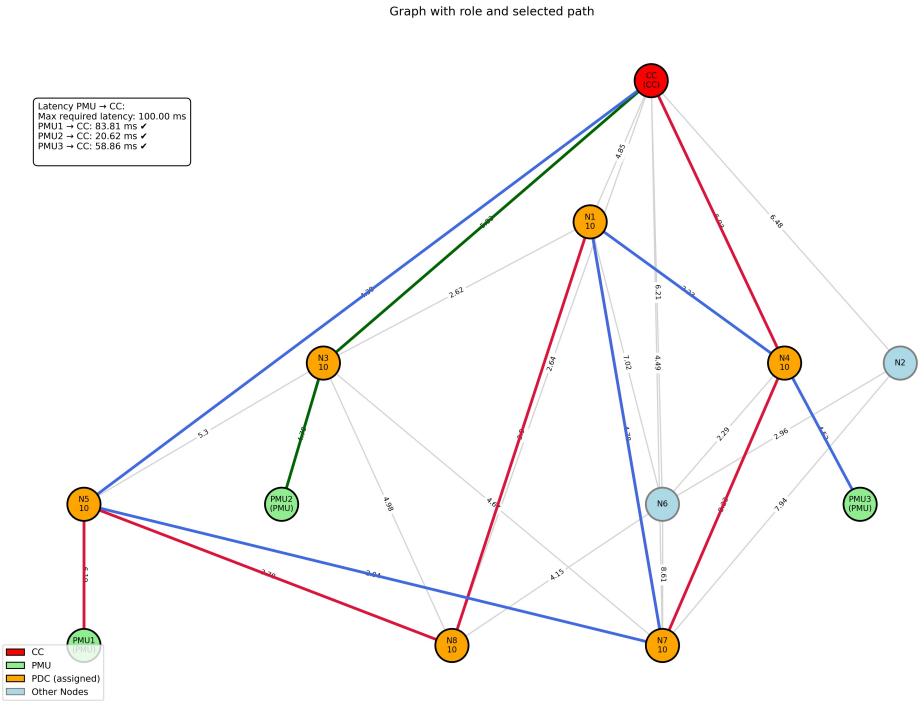

Figure 3.6: Random with splitting enable

rules, it can occasionally discover feasible configurations that are missed by the greedy strategy, especially in irregular or highly asymmetric topologies. However, the absence of an explicit optimization criterion frequently results in inefficient placements, characterized by an excessive number of PDCs, redundant paths, and strong convergence of multiple PMUs toward the same intermediate nodes. This behavior increases the risk of bandwidth saturation and leads to poor resource utilization. Although the random strategy is useful as an exploratory baseline and for stress-testing the system under diverse routing patterns, it is not suitable for performance-critical or large-scale deployments.

In summary, the Brute Force algorithm guarantees optimality but lacks scalability; the Greedy algorithm offers excellent performance and scalability at the cost of global optimality; and the Random algorithm provides exploratory diversity but suffers from inefficiency and instability. Together, these three approaches provide a comprehensive perspective on the trade-offs between optimality, execution speed, and robustness in the problem of PDC placement.

Table 3.1: Comparison of the three PDC placement algorithms

Algorithm	**Strengths**	**Weaknesses**	**Recommended Use**
Brute Force	• Always finds the optimal configuration. • Serves as ground truth for comparison. • Evaluates all combinations.	• Extremely high computational cost. • Not scalable to large networks. • Requires full topology knowledge.	• Small or testbed graphs. • Validation and benchmarking.
Greedy	• Very fast and lightweight. • Reacts quickly to network changes. • Suitable for real-time adaptation.	• Considers only link latency (ignores processing time). • Can produce suboptimal global configurations.	• Real-time or iterative placement. • Dynamic networks.
Random	• Explores diverse configurations. • Can reveal alternative valid paths.	• No optimization criterion. • Tends to place too many PDCs. • Possible bandwidth saturation.	• Experimental exploration. • Sensitivity and stress testing.

Chapter 4

Manual Deployment

The main goal of this stage was to design and deploy a *functional prototype of the openPDC architecture on Kubernetes*, validating the interaction between its acquisition and aggregation layers.

The deployment was structured across two namespaces, lower and higher, each hosting a distinct level of PDC nodes. The lower-level PDC was connected to three PMUs generating the measurement streams. The PDC aggregates these measurements and forwards them to the higher-level PDC acting as central collector. This setup represented the minimal yet complete architecture required to demonstrate the data flow from PMU to central collector.

In this first phase, all configurations were performed manually through the openPDC Manager *graphical interface*, in order to understand the internal mechanisms of the system and verify its correct operation.However, the manual process quickly proved to be time-consuming, error-prone, and difficult to reproduce due to the complexity and limited reliability of the GUI.

4.1 First system structure and data flow

The deployed system was organized into two separate Kubernetes namespaces, named lower and higher, representing the *two hierarchical layers* of the openPDC architecture. Each namespace hosts a complete and independent instance of the processing chain, including a Phasor Data Concentrator (PDC), a Percona XtraDB Cluster database, and the related management services, while maintaining network connectivity across layers for data exchange.

This separation into namespaces allowed the architecture to simulate the two-tier aggregation model typical of wide-area monitoring systems: the lower-level PDCs collect and preprocess data from field PMUs, whereas the higher-level PDC acts as a central collector (CC), aggregating and storing data coming from multiple

subordinate nodes.

4.1.1 Lower namespaces

The lower namespace contains the following main components:

- *PMUs* (Phasor Measurement Units): Three pods (pmu-1, pmu-2, pmu-3), each representing a simulated PMU, continuously generate synchrophasor measurements and transmit them using the IEEE C37.118 protocol over TCP (port 4712).
- *Lower-level PDC* (openpdc-lower): A PDC instance responsible for receiving and time-aligning the PMU data streams. It aggregates measurements into coherent data frames and exposes them through configurable OutputStreams, which define the destination and data content to be sent to other PDCs or to historian services.
- *Database (Percona XtraDB Cluster)*: A high-availability database system used by openPDC to store configuration, measurement metadata, and historian data. The cluster runs as a Percona XtraDB Cluster (PXC) managed by the Percona Operator for Kubernetes. In this setup, the pods cluster1-pxc-0, cluster1-haproxy-0, and the operator (percona-xtradb-cluster-operator) ensure consistency and replication.
  - *PXC (Percona XtraDB Cluster)* is a MySQL-compatible multi-master cluster based on Galera replication, providing synchronous data consistency across nodes.
  - *The HAProxy service (cluster1-haproxy and cluster1-haproxy-replicas)* acts as a load balancer, routing SQL queries to the healthy PXC nodes and exposing the database on standard ports (3306, 33060).
- *Services (Kubernetes)*: NodePort and ClusterIP services expose the database and PDC endpoints, allowing both internal communication and external access for testing. For example, the service openpdc-low exposes ports 4712 (C37.118 data input), 8500 (Web UI), and 6165 (command/control).

The lower namespace, therefore, represents the field level of the system, where PMUs continuously send raw measurement data to the PDC for local aggregation and preliminary synchronization.

4.1.2 Higher namespace

The higher namespace mirrors the same structure, with its own instance of openpdc-higher and a corresponding Percona XtraDB Cluster. Its role is to act as the central

aggregation layer (Control Center), receiving pre-processed data from one or more lower-level PDCs. The incoming data streams are handled as InputStreams, which feed into the historian, a database component that stores long-term measurements for analysis and visualization. In this configuration, the higher PDC thus operates both as a collector and a historical data repository, enabling centralized access to system-wide information. The historian is implemented within the same Percona database, which maintains tables for measurement definitions, timestamps, and device hierarchies.

4.2 Logical architecture and data flow

Each Phasor Measurement Unit (PMU) continuously generates *time-synchronized electrical measurements* (phasors, frequency, and rate of change of frequency) and transmits them in real time over TCP using the IEEE C37.118 protocol. These measurements are sent to the lower-level PDC, which serves as the first *aggregation point* in the network. Upon reception, the lower PDC aligns the incoming data streams based on their timestamps, validates their integrity, and aggregates them into coherent frames. The PDC thus ensures that all signals belonging to the same time instant are grouped together, allowing a consistent and synchronized view of the system state. Once the aggregation process is complete, the lower PDC prepares the data for transmission through a dedicated OutputStream , a configurable data channel that specifies which measurements to forward.

The higher PDC is configured to *receive* the data as an input stream, processes it in the same way as the lower one (time alignment and validation), and stores it in its Historian, a dedicated database component responsible for archiving and organizing all collected measurements. The historian enables the retrieval and visualization of historical data and serves as the long-term storage layer of the system.

In this configuration, the higher-level PDC acts as a central node, collecting the aggregated outputs of one or more lower PDCs. This setup represents the *minimal working scheme* of the overall architecture, where data flow seamlessly from acquisition (PMUs) to intermediate processing (lower PDC) and finally to central storage (higher PDC).

Although the current implementation involves only two layers, the same structure can be extended to build more complex hierarchies. By deploying additional namespaces and interconnecting multiple PDCs through OutputStreams, the architecture can scale to multi-level processing chains while maintaining the same configuration logic at each layer.

4.3 Initial manual deployment using openPDC Manager

The first deployment phase was carried out manually through the openPDC Manager graphical interface, with the goal of creating a minimal but functional topology composed of one lower-level PDC, one higher-level PDC, and three PMUs. The entire infrastructure was deployed on a lightweight Kubernetes distribution based on *k3s*, selected for its suitability in edge-oriented and resource-constrained environments. This configuration was designed to verify correct end-to-end communication between measurement devices and concentrators, and to gain a detailed understanding of how openPDC manages connections, data aggregation, and storage within a containerized orchestration platform.

Since the openPDC Manager application is only available for Windows, while the k3s cluster was running on a Linux-based host, it was necessary to configure *port forwarding* between the local Windows machine and the corresponding Kubernetes services. This was achieved by forwarding the management and data ports of each PDC instance (for example, 4712 for C37.118 data, 8500 for the web interface, and 6165 for command/control). Through this configuration, openPDC Manager was able to connect to the PDC instances running inside the k3s cluster as if they were local servers, allowing direct management of devices, output streams, and historians.

The manual configuration process involved *several steps*: first, adding each PMU to the lower-level PDC by specifying its IP address and protocol; then defining the corresponding OutputStream to forward the aggregated data to the higher-level PDC; and finally configuring the higher-level PDC to receive the data and store them through a properly initialized Historian. This approach made it possible to successfully validate the logical flow of data from PMUs to the central collector.

However, while this method provided valuable insight into the inner workings of the platform, it also revealed several limitations. The graphical interface of openPDC Manager proved to be *non-intuitive and error-prone*, often requiring multiple steps for relatively simple operations. Moreover, being a graphical interface, it cannot be used for automated configuration procedures. This limitation made it difficult to reproduce configurations consistently and highlighted a lack of scalability: each new PDC or PMU had to be added manually, making the process impractical for larger deployments.

For these reasons, the manual setup was considered a proof of concept, useful for understanding system behavior but unsuitable for repeated or large-scale use. This motivated the development of an automated solution capable of replicating the same configuration logic directly at the database level, ensuring both consistency and scalability in future deployments.

Chapter 5

Automation through CLI

The manual deployment described in the previous chapter successfully validated the logical architecture of the monitoring system and clarified the internal configuration mechanisms of openPDC. However, it also exposed several structural *limitations of the GUI-based approach*, including poor scalability, lack of reproducibility, and operational fragility when managing multiple interconnected components.

In particular, the dependence on the openPDC Manager graphical interface made configuration time-consuming and error-prone, while offering limited control over the underlying database operations. As the number of PDCs and PMUs increases, manual configuration becomes impractical and incompatible with automated or large-scale deployment scenarios.

For these reasons, the focus shifts in this chapter toward the development of a *command-line automation strategy* capable of reproducing the same configuration logic in a deterministic and scalable manner. Instead of interacting with openPDC through its graphical interface, the proposed approach directly analyzes and manipulates the underlying database schema, enabling programmatic control over device registration, stream creation, historian configuration, and inter-PDC connections.

This transition marks a fundamental step in the evolution of the system: from a manually configured proof of concept to a reproducible and automation-ready deployment pipeline. The following sections detail the analysis of the openPDC database and the design of the CLI tool that encapsulates the extracted configuration logic.

5.1 Analysis of the openPDC database

After completing the manual deployment through the graphical interface, the next objective was to analyze the internal behavior of openPDC during configuration, in order to reproduce those same operations automatically. Since all configuration

data, including PMU definitions, PDC parameters, OutputStreams, and Historian settings, are persisted in a relational database, understanding the *structure and interactions within the Percona XtraDB Cluster* became a fundamental step toward automation.

The database used by openPDC follows a MySQL-compatible schema and stores both static and dynamic information about devices, measurements, and processing chains. Each entity (e.g., a PMU, an OutputStream, or a Historian) corresponds to one or more tables, linked through foreign keys and internal identifiers. For instance, creating a new PMU involves inserting a record in the Device table, defining its corresponding measurements in the Measurement table, and linking them through an entry in SignalType. Similarly, creating an OutputStream or establishing a connection between two PDCs modifies several relational tables simultaneously.

To identify the exact sequence of queries executed by openPDC Manager during these operations, the MySQL general_log was enabled within the Percona pod. This log records every SQL statement executed on the server, including those generated by the openPDC application. By temporarily activating it during specific GUI actions, it was possible to isolate and observe the complete set of database queries associated with each configuration step. The procedure followed was the following:

1. Enable query logging by setting log_output='TABLE' and general_log=ON within the Percona container.

2. Perform a single configuration operation from the openPDC Manager GUI (e.g., add a PMU, create an OutputStream, configure a Historian, or connect two PDCs).

3. Disable logging immediately afterward with SET GLOBAL general_log=OFF.

4. Inspect the table mysql.general_log to extract and analyze all SQL statements executed during that time window.

By filtering out read-only queries (such as SELECT statements used by the GUI for status checks) and focusing on INSERT, UPDATE, and DELETE statements, the relevant modification queries were identified. This allowed a precise reconstruction of the configuration logic used internally by openPDC Manager. Each action could then be represented as a deterministic sequence of *SQL commands* directly executable inside the database pod.

This analysis revealed not only which tables were involved, but also the order of dependencies among entities: for example, a PMU must exist before creating its measurements, and an OutputStream can only be defined once its source device is registered. It also clarified the role of internal identifiers (such as ApplicationRoleID, UserAccountID, and NodeID), which are automatically generated and linked across

tables. Understanding these relationships was essential to reproduce the same behavior programmatically without relying on the GUI.

The knowledge gained through this analysis formed the foundation for developing an automated command-line tool (CLI) capable of executing equivalent database operations in a controlled and repeatable manner. By encapsulating the necessary SQL logic into modular commands, it became possible to create, configure, and interconnect multiple PDC instances automatically, eliminating the need for manual interaction with openPDC Manager while preserving the same functional outcome.

5.2 Configuration automation via CLI

To overcome the limitations of the GUI-based setup described in the previous chapter, the configuration process was automated through a Bash-based Command-Line Interface (CLI). The objective of this tool is to reproduce, in a deterministic and repeatable way, the same configuration steps previously executed through the openPDC Manager graphical interface.

The CLI executes predefined SQL sequences directly inside the Percona XtraDB Cluster pod and, after updating the database, triggers a configuration reload in the corresponding openPDC instance. This ensures that configuration changes take effect immediately without requiring manual interaction or service restarts.

The tool is implemented as a single script organized around multiple subcommands. Each subcommand encapsulates a specific configuration task (e.g., adding a PMU, creating an OutputStream, registering a Historian, or linking two PDCs), while sharing a common execution framework characterized by the following design principles:

- *Shared validation layer.* All global parameters (namespace, database name, cluster prefix, PXC pod, HAProxy service, secret name, openPDC pod) are validated before execution. The script verifies the existence of the required Kubernetes resources and retrieves database credentials from Kubernetes secrets. SQL statements are executed non-interactively inside the database pod using `kubectl exec`, ensuring compatibility with automated pipelines.
- *Safety and usability.* The script runs with strict error handling, provides structured help messages, and prints concise status information for traceability. Whenever possible, secondary parameters are derived automatically from user inputs to reduce configuration errors.
- *Idempotency.* Before inserting new entities, the CLI checks for their prior existence in the database and uses stable identifiers to prevent duplication. This guarantees that repeated executions of the same command produce predictable and consistent results.

- *Controlled runtime assumptions.* The tool assumes that the database is accessed through an HAProxy service, that openPDC runs inside a dedicated screen session within the pod, and that naming conventions follow a predefined cluster prefix unless explicitly overridden.

In the following subsections, each subcommand is described in detail, outlining its purpose, required parameters, and internal logic.

5.2.1 createaccount command

Purpose. Prepare application-level authentication inside openPDC's configuration database by ensuring the presence of standard roles and creating a named user bound to the `Administrator` role.

Key inputs.

- `-username`
- `-firstname`
- `-lastname`
- `-password`
- Global options (Kubernetes contexts, namespaces, database, and pod identifiers)

Logic.

1. Ensure standard roles `Administrator`, `Editor`, `Viewer` exist (idempotent inserts bound to the local `NodeID`).
2. Insert a `UserAccount` with database-backed authentication (`UseADAuthentication=0`), using a pre-hashed password value in the script (stored in `PHESC`).
3. Bind the user to the `Administrator` role via `ApplicationRoleUserAccount`, guarding against duplicates.

Notes.

- The script currently embeds a placeholder password hash; in production, pass a real hash derived from the clear-text password (or generate it server-side with a controlled routine) to avoid storing secrets in the script history.
- After database writes, `ReloadConfig` is sent to the openPDC pod so that changes take effect without restarting services.

5.2.2 createhistorian command

Purpose. Register a `LocalOutputAdapter` historian entry that `openPDC` uses for measurement archiving.

Key inputs.

- `-name` (optional, default: `localhistorian`)
- `-acronym` (optional, default: `LOCAL`)
- `-connstr` (optional, nullable)
- `-mri` (optional, default: `100000` µs)
- Global options (Kubernetes contexts, namespaces, database, and pod identifiers)

Logic.

1. Resolve `@NodeID` and perform an `INSERT` into the `Historian` table using:
   - Assembly: `HistorianAdapters.dll`
   - Type: `HistorianAdapters.LocalOutputAdapter`
   - Parameters: `IsLocal=1` and the selected `MeasurementReportingInterval`
2. Reload the `openPDC` configuration to apply the new historian entry.

Notes.

- Keeping the historian local simplifies deployment and reduces moving parts. If needed, a remote historian can be introduced by changing the adapter type and connection string.
- The script prints a clear summary of the historian identity before applying changes, ensuring transparency and reproducibility of the configuration process.

5.2.3 addpmu command

Purpose. Create a PMU device and its associated measurements/phasors in the lower layer database.

Key inputs.

- `-name` and/or `-acronym`
- Optional: `-server`, `-port` (default: `4712`), `-fps` (default: `25`)
- When `-acronym` or `-server` are omitted, they are automatically derived from `-name`.
- Global options (Kubernetes contexts, namespaces, database, and pod identifiers)

Logic.

1. Resolve `@NodeID`, allocate `@UniqueID`, and derive the next `AccessID` from the numeric suffix of the PMU acronym.
2. Insert into `Device` with `IsConcentrator=0`, and define a connection string that encodes the transport protocol (TCP), the server endpoint, and the listener mode (`islistener=false`). Default coordinates and timing parameters are set.
3. Insert the core measurements (analog value, frequency, df/dt, status flags) with canonical `SignalTypeID` values and point tags (e.g., `_ACRONYM:F`, `_ACRONYM:DF`), plus the phasor set (A/B/C) with magnitude and phase-angle measurements and corresponding entries in the `Phasor` table.
4. Insert diagnostic and statistic measurements (e.g., `GPA_ACRONYM!PMU:ST*`, `GPA_ACRONYM!IS:ST*`) to monitor device and input-stream health and counters.
5. Trigger `ReloadConfig` on the `openPDC` pod to apply the configuration changes.

Notes.

- The command prints a pre-flight summary and refuses to proceed if a device with the same `Acronym` already exists (soft idempotency check).
- Default parameters (FPS, coordinates, connection options) can be refined later; keeping them explicit ensures reproducibility of each configuration run.

5.2.4 createoutputstream command

Purpose. Build a unidirectional `OutputStream` on the lower PDC that forwards a selectable set of PMU signals.

Key inputs.

- `-name`, `-acronym`
- `-pmus` (comma-separated list, e.g., `"PMU-1,PMU-2,PMU-3"`)
- Optional parameters:
  - `-port` (command/data channel; default: `4712`)
  - `-fps` (default: `30`)
  - `-nomfreq` (default: `60`)
  - `-lag`, `-lead`, and `-user`
- Global options (Kubernetes contexts, namespaces, database, and pod identifiers)

Logic.

1. Insert an `OutputStream` row (adapter) defining timing (`Lag/Lead`), frame rate, nominal frequency, data formats, and the connection string (port configuration).
2. For each PMU specified in `-pmus`:
   (a) Insert an `OutputStreamDevice` entry and its A/B/C phasor descriptors.
   (b) Populate `OutputStreamMeasurement` by selecting canonical measurement IDs from `Measurement` joined with `Device`, including analog value, frequency, df/dt, status, and the phasor magnitude/angle pairs (A/B/C).
3. Reload the `openPDC` configuration to apply changes.

Notes.

- The stream definition is declarative: once PMUs are mapped, the adapter publishes aligned frames to the remote endpoint configured at the receiving side.
- The command prints the list of PMUs bound to the stream, improving traceability and auditability.

5.2.5 connectiontopdc command

Purpose. Configure the link from a lower PDC to a higher PDC by creating a concentrator device pointing to the remote server and by registering PMU children under it, so that the lower layer exports the desired signals upstream.

Key inputs.

- `-name`, `-acronym`
- `-server` (host of the target PDC)
- `-pmus` (comma-separated list of PMUs to attach)
- `-port`
- Global options (Kubernetes contexts, namespaces, database, and pod identifiers)

Logic.

1. Update the `Node.Settings` field with endpoints related to the remote status server and data publisher for the target `-server` (web/status/historian URLs and TCP ports).
2. Create a parent `Device` row with `IsConcentrator=1` and a connection string targeting `-server` at the port specified through `-port`.
3. For each PMU in `-pmus`, insert a child `Device` entry under the parent and add the core measurements (F, DF, S) along with the A/B/C phasor magnitude and phase measurements.
4. Reload the `openPDC` configuration to activate the new hierarchy.

Notes.

- This routine codifies the hierarchical topology directly in the database schema (parent/child `Device` entries), mirroring the behavior of the GUI during inter-PDC linking.
- The script assigns stable `LoadOrder` and `AccessID` values to maintain deterministic ordering of devices within each PDC.

5.2.6 Command-line configuration workflow

The combination of the previously described subcommands allows the complete configuration of a PDC instance directly from the command line, without the need to rely on the graphical interface of openPDC Manager. Each command encapsulates a specific step of the setup process, from user account creation to historian registration, PMU insertion, stream definition, and inter-PDC connection,

reproducing the same logic that would otherwise require multiple manual operations through the GUI.

This approach provides a series of practical advantages:

- *Simplicity and reproducibility.* Once the basic parameters (namespace, database pod names) are known, configuring a new PDC requires only a few well-structured commands. The same setup can be repeated across environments or namespaces with deterministic results.
- *Automation and scalability.* The process can be embedded in deployment scripts or pipelines, enabling large-scale or hierarchical deployments without manual intervention.
- *Transparency and auditability.* Each SQL operation is explicit and traceable, and the CLI prints clear summaries before applying any change. This ensures visibility of the configuration state and facilitates debugging.
- *Immediate effect.* The automated `ReloadConfig` after each operation allows openPDC to load the new configuration on the fly, avoiding service restarts and minimizing downtime.

Through this workflow, a complete *openPDC environment*, including user management, measurement devices, output streams, historians, and inter-PDC links, can be provisioned automatically within a Kubernetes namespace. The resulting system is not only easier to maintain but also serves as a reproducible foundation for more complex multi-layer topologies, where each PDC can be instantiated and linked via script in a consistent and scalable manner.

The following listing shows the help output of the `openpdc_cli.sh` script, summarizing its subcommands and options.

```
Usage:
  ./openpdc_cli.sh <subcommand> [global options] [command options]

Subcommands:
  addpmu                Add a PMU (Device + Measurements)
  createoutputstream    Create an output stream
  createhistorian       Create a local historian
  connectiontopdc       Connect to a lower level PDC
  createaccount         Create a MySQL user account
  help                  Show this help

Global options:
  --db-context CONTEXT            Kubernetes context for the database
    cluster
```

```
  --openpdc-context CONTEXT       Kubernetes context for the openPDC
    cluster
  --db-ns NAMESPACE               Database namespace
  --pdc-ns NAMESPACE              openPDC namespace
  --db NAME                       Database name
  --pod NAME                      (e.g., openpdc-pod-1234-5678)
  --cluster-prefix NAME           (default: cluster1)
  --pxc-pod NAME                  (default: <cluster>-pxc-0)
  --haproxy-svc NAME              (default: <cluster>-haproxy)
  --secret-name NAME              (default: <cluster>-secrets)
  --mysql-user USER               (default: root)

addpmu options:
  --name NAME             (e.g., "Pmu-3") mandatory
  --acronym ACR           (e.g., PMU-3)
  --server HOST           PMU endpoint
  --port N                (default: 4712)
  --fps N                 (default: 25)

createoutputstream options:
  --acronym ACR               (e.g., LOWER) mandatory
  --name NAME                 (e.g., "low2high") mandatory
  --pmus "PMU-1,PMU-2,..."    PMUs to include in the stream
  --port N                    CommandChannel port (default: 4712)
  --fps N                     FramesPerSecond (default: 30)
  --nomfreq N                 NominalFrequency (default: 60)
  --lag MS                    LagTime in ms (default: 3)
  --lead MS                   LeadTime in ms (default: 1)
  --user TAG                  UpdatedBy/CreatedBy (default: polito)

createhistorian options:
  --name NAME                 (default: localhistorian)
  --acronym ACR               (default: LOCAL)
  --connstr STR               ConnectionString (default: NULL)
  --mri N                     MeasurementReportingInterval (default:
    100000)
  --user TAG                  UpdatedBy/CreatedBy (default: polito)

connectiontopdc options:
  --name NAME                 mandatory
  --acronym ACR               mandatory
  --server HOST               mandatory
  --pmus LIST                 mandatory

createaccount options:
  --username USER             mandatory
  --password PWD              mandatory
  --firstname NAME            mandatory
  --lastname NAME             mandatory
```

Chapter 6

Integration of placement algorithms within Kubernetes

The previous chapter introduced a command-line automation layer capable of reproducing the configuration logic of openPDC in a deterministic and scalable manner. By directly manipulating the underlying database schema, it became possible to eliminate manual interaction with the graphical interface and to ensure reproducibility of PDC setup within a Kubernetes environment.

However, while Chapter 5 focused on automating the configuration of individual PDC instances, it did not address the integration between the *placement decision logic* and the *orchestration infrastructure* responsible for deploying and managing multiple clusters. This chapter introduces the integration between algorithmic placement strategies and real Kubernetes-based deployment. The goal is to translate placement decisions, expressed as graph paths from PMUs to the Control Center, into a concrete multi-cluster infrastructure automatically deployed and configured. To achieve this, a fully automated pipeline interprets the output of the placement algorithms, provisions the required clusters, deploys the PDC and PMU components, and applies the corresponding hierarchical configuration.

Through this integration, placement decisions become actionable deployment instructions executed within a real Kubernetes environment..

6.1 Automation Pipeline

The integration between placement algorithms and Kubernetes orchestration is implemented through a structured automation pipeline. Rather than treating

placement, deployment, and configuration as separate phases, the system organizes them into a coordinated workflow that transforms an abstract placement decision into a fully deployed and operational *multi-cluster infrastructure*.

The pipeline is composed of three complementary components:

- `autopdc_configurator.py`, responsible for executing the selected placement algorithm and generating a structured representation of the resulting PMU–PDC–CC paths;
- `deployer.sh`, responsible for provisioning the required Kubernetes clusters and deploying the corresponding PDC and PMU instances;
- `applier.py`, responsible for configuring the logical relationships among deployed PDCs according to the hierarchical paths defined by the placement solution.

Together, these components implement a deterministic transformation from a graph-based optimization output to a concrete, runnable infrastructure. This modular design separates decision logic from infrastructure management and configuration concerns, while ensuring that the entire process can be executed automatically and repeated consistently whenever topology changes require a new placement.

6.1.1 Entry point: autopdc_configurator.py

The first script in the pipeline, autopdc_configurator.py, acts as the entry point for the entire process. It drives the selection of the *PDC placement algorithm* and the modification of parameters such as edge status, latency, and bandwidth, which in turn produces an output.json file. This file contains, for each PMU, the path connecting it to the final controller node (CC) through the selected PDCs. Its structure is as follows:

```
{
    "path": {
        "PMU1": {
            "path": [ "PMU1","N4", "N14", "CC" ],
            "delay": 48.86
        },
        "PMU2": {
            "path": [ "PMU2", "N13", "CC" ],
            "delay": 25.69
        },
    },
}
```

The script parses this structure and makes it available to the other components in the pipeline. Once the JSON file has been loaded, autopdc_configurator.py sequentially triggers the next two scripts: deployer.sh to create the infrastructure and applier.py to configure the PDCs. This design separates the deployment and configuration phases, ensuring modularity and facilitating future extensions of the system.

6.1.2 Infrastructure Deployment: deployer.sh

The second component of the pipeline, deployer.sh, is responsible for provisioning the *entire infrastructure*. Its main task is to analyze the path structure and dynamically create:

- the Kubernetes clusters required to host the PDCs: all clusters are created using k3d, which provides lightweight Kubernetes clusters by running k3s in Docker containers. Since k3s is containerized in this setup, deployer.sh explicitly exposes and maps the network ports used by the PDCs at cluster creation time. This port mapping is necessary to enable direct communication between PDCs hosted in different clusters, allowing inter-cluster data streams to be established despite the containerized nature of the Kubernetes control planes.
- the PDC Pods within the appropriate clusters according to the defined topology: one pdc for each cluster;
- the simulated PMUs, deployed in the same cluster as the closest PDC in the computed path. These components are introduced solely for testing purposes, in order to generate monitoring data for the system.
- an additional cluster dedicated to the PerconaXtraDB database (used by openPDC).

A crucial part of this process is the *initialization of the Percona database*. To allow each PDC to authenticate and create its internal data structures, the script copies the Percona DB secret into every PDC cluster using kubectl apply. The PDC init-container uses this secret to configure database access and generate the necessary tables. Without this step, PDCs would not be able to complete their startup phase and reach the Ready state. The script is also designed to fully recreate the environment when necessary. If autopdc_configurator.py detects a change in the input JSON, deployer.sh removes all PDC Pods and clears the database volume, guaranteeing that the infrastructure is regenerated coherently with the new configuration.

6.1.3 PDC Configuration: applier.py

The third script in the pipeline, `applier.py`, is responsible for the most delicate phase of the system: the logical configuration of the PDCs and the establishment of the hierarchical dependency chains among them. Starting from the JSON file generated in the previous steps of the pipeline, the script translates an abstract description of the PMU–PDC–CC paths into a concrete and consistent *openPDC configuration* deployed across multiple Kubernetes clusters.

For each PDC, the script automatically generates the complete set of configuration entities required by openPDC, including historians, PMU definitions, downstream and upstream connection strings, acronyms, listening ports, and output streams. In parallel, it reconstructs the logical data-flow topology, explicitly encoding the aggregation chains of the form PMU $\rightarrow$ PDC $\rightarrow \cdots \rightarrow$ CC. This dual construction ensures that both the local configuration of each PDC and the global structure of the monitoring infrastructure remain aligned with the placement solution produced by the optimization algorithms.

Since PDCs are organized in hierarchical chains, their configuration is subject to strict dependency constraints: a PDC cannot be fully configured unless all the PDCs it depends on are already deployed and operational. To address this issue, `applier.py` builds an explicit dependency graph among clusters and computes a valid execution order through a topological sorting procedure. This guarantees a safe and deterministic configuration process, preventing race conditions, partial configurations, and inconsistent states. In practice, configuration proceeds bottom-up along the aggregation hierarchy, starting from the PDCs closest to the PMUs and progressively moving toward higher-level PDCs and the final collection point.

Before applying any configuration step, the script performs a series of *validation and synchronization checks*. These include verifying the existence of the referenced Kubernetes contexts, waiting for the openPDC pods to reach the `Running` state, and ensuring that all required dependencies are available. Additionally, the script enforces consistent naming and acronym generation to avoid collisions, and it prevents duplicate or premature configuration commands that could lead to undefined behavior in openPDC.

The actual configuration is executed by invoking the openPDC command-line interface directly inside the appropriate Kubernetes pods. This approach allows the script to act as an automated *configuration applier*, bridging the gap between the abstract placement model and a fully operational multi-cluster deployment. By encapsulating all configuration logic into a single reproducible procedure, `applier.py` significantly reduces manual intervention, improves reliability, and enables systematic experimental validation of different PDC placement strategies on real Kubernetes-based infrastructures.

6.2 Automatic Regeneration of the Environment

An important aspect of the architecture is the handling of configuration changes. When the user modifies the output.json file and relaunches autopdc_configurator.py, the pipeline performs the following actions:

1. Cleans the Percona database, removing outdated tables and configurations;
2. Deletes all PDC Pods, forcing a full re-initialization through their init-container. This step ensures that the PDC instances regenerate their configuration tables in the database, which was cleaned in the previous step. Without restarting the pods, already running PDC instances would not reinitialize their schema and therefore the required tables would not be created;
3. Recreates all PDCs and PMUs according to the updated topology;
4. Applies the new configuration through applier.py.

This ensures that the entire system can be regenerated deterministically and consistently with the new PDC placement, without requiring manual intervention and keeping the pipeline fully automated.

6.2.1 Computing Continuum Perspective

This regeneration capability is particularly relevant in the context of the *computing continuum*, where computational resources are distributed across heterogeneous and geographically dispersed environments. In such scenarios, infrastructure conditions may change dynamically due to variations in network latency, bandwidth availability, node status, or workload distribution.

By combining automated cluster provisioning, database re-initialization through init-containers, and deterministic configuration replay, the proposed pipeline enables a *closed-loop adaptation mechanism* between placement decisions and runtime infrastructure state. The system can be reconfigured from scratch in a controlled and repeatable manner, ensuring that placement decisions remain continuously aligned with current network conditions.

This behavior reflects a fundamental characteristic of computing continuum architectures: the ability to dynamically orchestrate, redeploy, and recompose distributed resources across multiple administrative domains without manual intervention. In this sense, the automatic regeneration mechanism does not merely simplify deployment, but acts as a foundational enabler for adaptive, latency-aware multi-cluster orchestration.

Chapter 7

Testing and final evaluation

This chapter presents the testing methodology and the final experimental evaluation of the proposed PDC placement strategies. The objective of this evaluation is not limited to the theoretical analysis of the algorithms, but extends to the assessment of their behavior when deployed in a realistic and dynamic operational environment.

The experiments are designed to validate the end-to-end workflow introduced in the previous chapters, encompassing placement computation, infrastructure provisioning, deployment of PMU and PDC components, and configuration of data flows. Particular attention is given to the impact of dynamic network conditions, in order to evaluate how effectively the system adapts to changes and how these adaptations translate into real execution overhead.

The chapter first introduces the objectives of the experimental evaluation and the characteristics of the testbed and automation framework. It then defines the evaluation metrics adopted to analyze temporal performance, topological stability, and scalability. Finally, the experimental results are presented and discussed, providing a comparative analysis of the considered placement algorithms and highlighting their strengths and limitations in practical deployment scenarios.

7.1 Objectives of the Experimental Evaluation

The objective of this experimental evaluation is to systematically assess the behavior and performance of different PDC placement algorithms in *dynamic network scenarios*. Unlike evaluations that focus exclusively on algorithmic complexity, this work considers the end-to-end execution time required to adapt the system to changing network conditions.

In particular, the evaluation explicitly accounts for the time required to create clusters, deploy PMU and PDC components, and configure the deployed instances, in addition to the time spent by the placement algorithms to compute a new

deployment strategy. This holistic approach allows the assessment of the practical overhead introduced by each solution when applied in a realistic operational environment.

A key objective of the experiments is the analysis of *temporal performance*, achieved by decomposing the overall execution time into distinct phases: placement computation, infrastructure provisioning and deployment, and PDC configuration. This decomposition enables a clear distinction between algorithmic decision-making costs and infrastructure-related delays, which are often dominant in real-world distributed systems.

Another objective of the evaluation is to investigate the *topological stability* of the solutions produced by the different algorithms. In dynamic scenarios, changes in network conditions may trigger repeated reconfigurations; therefore, it is important to quantify how much the set of deployed PDCs varies across successive iterations. Evaluating the stability of placement decisions provides insights into the ability of each algorithm to minimize unnecessary redeployments while still adapting to network changes.

Finally, the experimental evaluation aims to assess the *scalability of the placement algorithms* with respect to both the network size and the number of PMUs. By progressively increasing the number of nodes and data sources, the experiments analyze how the computational cost of the placement phase evolves, while also observing the impact on deployment and provisioning times.

Overall, this evaluation provides a comprehensive comparison of the considered placement strategies, capturing both algorithmic efficiency and operational overhead, and offering insights into their suitability for deployment in dynamic computing continuum environments.

7.2 Testbed and Experiment Automation

This section describes the experimental infrastructure and the automation framework adopted to execute the evaluation campaigns. The objective is to provide a precise description of the testbed configuration and the mechanisms used to coordinate placement computation, deployment procedures, topology modifications, and metric collection.

The experimental workflow combines two complementary elements: a parameterized test scenario defining the network topology and placement constraints, and an automation layer responsible for orchestrating repeated executions in a controlled and reproducible manner.

The following subsections detail the experimental environment, the automation process, and the characteristics of the generated test scenarios.

7.2.1 Experimental Environment

All experiments were conducted on a dedicated single-node server. The hardware configuration is summarized below:

- 2 × Intel® Xeon® Gold 6442Y (48 physical cores, 96 logical CPUs)
- 503 GB RAM
- 17 TB SSD storage
- 10 Gbps network interface card
- x86_64 architecture with 2 NUMA nodes

The software stack adopted for the experimental evaluation includes the operating system, the containerization environment, the Kubernetes-based orchestration layer, and the application-level components required to deploy and manage the monitoring infrastructure:

- Ubuntu 24.04.1 LTS
- Docker v29.2.1
- k3d v5.8.3 (k3s v1.31.5-k3s1)
- Kubernetes server v1.29.4+k3s1
- openPDC v2.4
- Percona Operator v1.11.0

The adopted platform provides substantial computational and memory resources. The objective of the experimental evaluation is not to assess the feasibility of running the system on resource-constrained local machines, but rather to analyze the behavior of the placement strategies and the orchestration workflow in a controlled environment *without artificial hardware bottlenecks*. By operating on a sufficiently provisioned infrastructure, the results reflect the intrinsic algorithmic and orchestration-related overhead, rather than limitations imposed by the execution host.

7.2.2 Run Automation

The execution of the experimental campaigns is coordinated by a dedicated automation script that orchestrates placement computation, infrastructure provisioning, controlled topology updates, and metric collection in a fully reproducible manner.

The automation of the experimental workflow was necessary not only to guarantee repeatability, but also to address the significant execution time required by each test. Cluster creation, deployment of PMU and PDC components, configuration procedures, and repeated reconfigurations under dynamic conditions introduce non-negligible delays. Performing these operations manually would be impractical and highly error-prone, especially considering the number of runs required for statistical evaluation.

The automation layer interacts with the deployment configurator module (`deploy_automation.autopdc_configurator`) through a controlled execution flow. For each experimental run, the script:

- generates the network graph according to the specified parameters;
- sequentially executes the selected placement algorithms (Brute Force, Greedy, Random);
- optionally applies topology modifications between iterations;
- triggers deployment and configuration procedures;
- collects runtime measurements and deployment snapshots.

All applied topology updates are recorded and can be reverted through an automated restoration mechanism, ensuring that each algorithm is evaluated under identical initial conditions. The framework also manages environment cleanup between executions, preventing residual state from influencing subsequent runs.

Runtime information and intermediate results are stored in structured output directories, including execution time logs (`runtime.csv`) and deployment snapshots (`snapshot_XXXX.json`), enabling post-processing and statistical analysis.

This design guarantees repeatability, eliminates manual intervention, and ensures fair comparison across algorithms and experimental configurations.

7.2.3 Test Scenario

The experimental test scenario emulates a distributed synchrophasor monitoring environment in which multiple PMUs transmit measurements toward a Control Center (CC) through one or more intermediate PDC nodes. The communication infrastructure is modeled as a graph, where nodes represent PMUs, candidate

hosting locations for PDCs, and the CC, while edges represent network links characterized by latency, bandwidth, and availability status.

The network graph is generated automatically at the beginning of each run by the deployment configurator module (`deploy_automation.autopdc_configurator`). The topology generation is controlled through the following parameters: number of candidate nodes (`-num-candidates`), number of PMUs (`-num-pmus`), probability of additional links (`-p-extra`), minimum number of links incident to the CC (`-cc-min-links`), and number of links per PMU (`-pmu-links`).

All placement algorithms are evaluated under the same global constraint on end-to-end latency, defined as a maximum admissible delay of $L_{\max} = 80$ ms. Cluster splitting is disabled in all experiments (`splitting = n`), so that placement decisions are computed under a unified connectivity assumption.

Two experimental configurations are considered.

- **Dynamic conditions (Mode 1: topology changes).** This configuration evaluates the ability of the placement strategies to adapt to *changing network conditions*. Each run consists of a baseline topology state (T0) followed by successive modified states (T1–T4). For each algorithm, the placement and reconfiguration workflow is re-executed after every change. In the current setup, the automation is configured with `RUNS = 3`, `TS_PER_RUN = 5` (T0–T4), and `CHANGES_PER_T = 1`.

  At each topology step, the automation selects an edge at random among those observed in the latest deployment snapshot and applies exactly one modification, chosen randomly among: (i) *latency update*, where the new value is sampled uniformly in [10,25] ms; (ii) *bandwidth update*, where the new value is sampled uniformly in [50,300]; and (iii) *status update*, where the link is set according to the configured choices (in the current setup, `down`).

  To ensure fair comparison, after completing the sequence of changes for one algorithm, the automation restores the baseline network state by applying the inverse of the modifications recorded in the snapshots (UNDO mechanism) before starting the next algorithm.

- **Scalability analysis (Mode 2: increasing topology size).** This configuration focuses on the computational scalability of the placement phase by progressively increasing the topology size across successive runs, without introducing intra-run changes. The evaluated sizes are defined by the following sequences: candidate nodes {10,20,30,40} and PMUs {1,2,3,4}. For each size configuration, the placement computation time is extracted from `runtime.csv`, while the number of selected PDC nodes is computed from the generated deployment snapshots.

7.3 Evaluetion Metrics

This section presents the metrics adopted to evaluate the behavior of the placement algorithms under the two experimental configurations introduced in Section 2.2. The selected metrics are designed to capture multiple and complementary dimensions of system performance, including execution time, stability of placement decisions in dynamic conditions, and scalability with respect to the size of the network. Together, these metrics provide a comprehensive view of both algorithmic behavior and operational overhead in realistic deployment scenarios.

7.3.1 Temporal Metrics

Temporal metrics are primarily employed in the first type of experiment, where controlled topological changes are introduced within each run. In this scenario, each change triggers a new placement computation and a subsequent *reconfiguration* of the deployed infrastructure. As a result, the overall reaction time of the system is influenced not only by the placement algorithm itself, but also by the time required to provision and configure the necessary components.

For each topological change and for each placement algorithm, the execution time is decomposed into *three distinct phases*. The first phase corresponds to the placement computation time, defined as the time required by the algorithm to determine the set of nodes on which PDCs should be deployed. The second phase corresponds to the deployment time, which includes cluster creation as well as the deployment of PMU and PDC components. The third phase corresponds to the configuration time, during which the deployed PDCs are configured to establish the required data flows.

This temporal decomposition is represented using stacked plots, where each column corresponds to the execution of a specific algorithm following a topological change, and each stacked segment represents one of the three execution phases. This visualization highlights the relative weight of algorithmic decision-making compared to infrastructure-related operations, and enables a direct comparison of the end-to-end cost associated with different placement strategies.

To account for variability across executions, the total execution time associated with each change is collected over *multiple runs* and aggregated using box plots. These plots represent the distribution of execution times for each algorithm and for each change, capturing dispersion, median values, and potential outliers. This representation allows the evaluation of both average behavior and temporal robustness under repeated executions.

7.3.2 Topological Stability Metrics

Topological stability metrics are used to evaluate how placement decisions evolve across successive topological changes. In dynamic environments, frequent recomputation of placement strategies may lead to *significant modifications* of the deployed infrastructure, potentially resulting in unnecessary redeployments and increased operational overhead. For this reason, it is important to quantify the degree of similarity between consecutive placement configurations.

To this end, the *Jaccard distance* is adopted as a measure of dissimilarity between two placement configurations. Given two sets A and B, representing the sets of nodes hosting PDCs before and after a topological change, the Jaccard distance is defined as:

$$d_J(A, B) = 1 - \frac{|A \cap B|}{|A \cup B|}$$

The metric ranges from 0 to 1, where a value of 0 indicates identical configurations, while a value of 1 indicates completely disjoint sets.

As an illustrative example, consider a placement configuration before a change where PDCs are deployed on nodes $A = \{n_1, n_2, n_3\}$, and a new configuration after the change where PDCs are deployed on nodes $B = \{n_2, n_3, n_4\}$. In this case, the intersection $A \cap B = \{n_2, n_3\}$ and the union $A \cup B = \{n_1, n_2, n_3, n_4\}$. The resulting Jaccard distance is therefore $d_J(A, B) = 1 - \frac{2}{4} = 0.5$, indicating a partial modification of the deployed PDC set.

In the experimental evaluation, the Jaccard distance is computed for each placement algorithm after every topological change, providing a quantitative measure of how disruptive the new placement is with respect to the previous configuration. Lower values correspond to more stable placement strategies, while higher values indicate more significant reconfiguration.

To obtain statistically meaningful insights, Jaccard distance values are collected over *multiple runs* and aggregated using box plots. These plots represent the distribution of the Jaccard distance for each algorithm and for each topological change, enabling a comparative analysis of placement stability across algorithms and repeated executions.

7.3.3 Scalability Metrics

Scalability metrics are employed in the second type of experiment, which focuses on analyzing the computational behavior of the placement algorithms as the size of the network increases. In this experimental configuration, no topological changes are introduced within a run. Instead, the *number of nodes and PMUs* in the network graph is progressively increased across successive runs, while keeping network conditions fixed within each run.

For each run and for each placement algorithm, the placement computation time is measured and analyzed as a function of the number of nodes in the graph, with a maximum runtime of one hour enforced for each execution, after which the execution is terminated. The resulting metrics are represented through plots where the number of nodes is reported on the x-axis and the placement time on the y-axis. This representation enables a direct comparison of scalability trends among algorithms, highlighting differences in growth rates and computational efficiency.

By isolating the placement phase from deployment and configuration overhead, these metrics provide a clear view of the intrinsic *algorithmic complexity* of the considered placement strategies, allowing the identification of scalability limits and practical applicability in large-scale scenarios.

In addition, an auxiliary plot is produced to relate the availability of routing alternatives to the placement outcome. In this plot, the x-axis reports the number of candidate nodes, while the y-axis reports the *number of selected PDC nodes* (including the CC) for each algorithm. This visualization helps assess whether, as the set of candidate nodes grows and more potential paths become available, the algorithms tend to select longer or shorter PDC chains. As a consequence, it supports the interpretation of the runtime results by clarifying whether variations in placement time are driven primarily by algorithmic complexity or are also influenced by the increased number of feasible paths and the resulting placement structure.

7.4 Experimental Results

This section reports the experimental results obtained from the evaluation experiments described in the previous sections. The results are organized to provide a structured analysis of the proposed placement strategies, starting from the validation of the experimental environment and progressively moving toward the discussion of algorithm behavior and comparative performance.

The section first presents a preliminary validation aimed at verifying the correctness of the experimental testbed, with particular focus on the enforcement of network latency. This validation step is necessary to ensure that the delays computed by the placement algorithms are effectively reproduced at runtime in the deployed infrastructure.

After this initial verification, the section analyzes the behavior of the different placement algorithms under dynamic network conditions, discussing their impact on deployment time, reconfiguration overhead, and topological stability. The final part of the section is devoted to the comparative analysis of the algorithms with respect to scalability, highlighting how their computational cost evolves as the size of the network increases.

7.4.1 Experimental Validation of Network Delay Enforcement

Before analyzing the experimental results, a **preliminary validation** was performed to verify the correctness of the network delay enforcement mechanism adopted in the testbed. In particular, this validation aimed to ensure that the latency values computed by the placement algorithms and derived from the network topology were effectively reflected in the runtime behavior of the deployed system.

To this end, **artificial delays** corresponding to the link latencies defined in the network graph were applied to the host network interfaces used by the `k3d` containers. These delays were injected at the host level, affecting the communication between the host and the containerized PMU and PDC instances, thereby emulating **realistic network conditions**.

In order to accurately measure the end-to-end latency toward the Control Center (CC), an additional PDC was deployed and directly connected to the CC. The latency was then measured on this final PDC through the OpenPDC Manager, ensuring that the observed value represented the full processing chain from the PMU to the CC-side aggregation point.

Experimental observations revealed that each PDC introduces a node processing delay of approximately 21.5 ms. Therefore, the total end-to-end latency must account not only for the sum of the link latencies along the path, but also for the cumulative processing contribution of each traversed PDC, including the additional PDC connected to the CC.

Formally, the total end-to-end latency can be expressed as:

$$L_{\text{total}} = \sum_{(u,v)\in\text{path}} L_{u,v} + N_{\text{PDC}} \cdot L_{\text{proc}} \tag{7.1}$$

where $L_{u,v}$ is the latency of link (u, v), N_{PDC} is the number of PDCs traversed along the path (including the one connected to the CC), and $L_{\text{proc}} \approx 21.5\,\text{ms}$ represents the average processing delay introduced by a single PDC.

The correctness of the applied delays was verified through the **OpenPDC Manager** by observing the end-to-end latency of data packets received by the final PDC. In order to obtain reliable measurements, multiple observations were performed under stable traffic conditions, ensuring that the system had reached a steady operational state before recording the latency values.

The measured latency was compared against the analytically computed value derived from the network topology and the selected placement configuration. In particular, the expected delay was computed according to Eq. 7.1, considering both link-level latencies and cumulative PDC processing contributions. The additional PDC connected to the CC was explicitly included in this computation.

Across the conducted tests, the observed latency values were consistently aligned with the theoretical expectations, with only negligible variations attributable to normal system scheduling effects and container-level overhead. This confirms that the delay injection mechanism effectively reproduces the modeled network conditions and that the processing contribution of each PDC (approximately 21.5 ms) is consistently reflected in the measured end-to-end delay.

The experimental setup used for this validation is illustrated in Fig. 7.1. The left side of the figure reports the logical topology employed during the validation phase, while the right side shows the latency values observed through the OpenPDC Manager at the PDC directly connected to the Control Center. This visual comparison highlights the correspondence between the modeled topology and the measured end-to-end delay, providing empirical confirmation of the correctness of the delay enforcement mechanism.

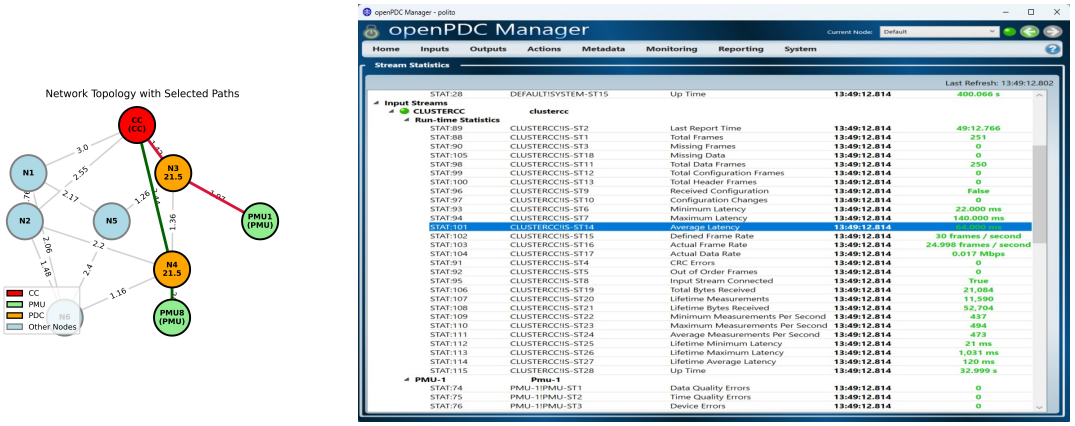

Figure 7.1: Experimental validation of network delay enforcement. The figure shows the network topology used for validation (left) and the corresponding end-to-end latency measured through the OpenPDC Manager (right).

This confirmation demonstrates that the experimental environment accurately enforces the modeled network latencies and processing overheads, and that the observed system behavior faithfully reflects the outcomes of the placement strategies.

7.4.2 Temporal Performance under Dynamic Conditions

Temporal performance under dynamic topology changes was evaluated by measuring the end-to-end execution time of the three placement algorithms across

successive topology indices (T0–T4). The total reaction time includes three components: *placement computation*, *infrastructure deployment* (deployer), and *PDC configuration* (applier).

Figure 7.2 presents the decomposition of execution time within a single experimental run. A first and clear observation is that the deployment phase dominates the overall execution time for all algorithms. The deployer component consistently represents the largest portion of the stacked bars, confirming that infrastructure provisioning and cluster-level operations constitute the primary operational overhead in the adaptive workflow.

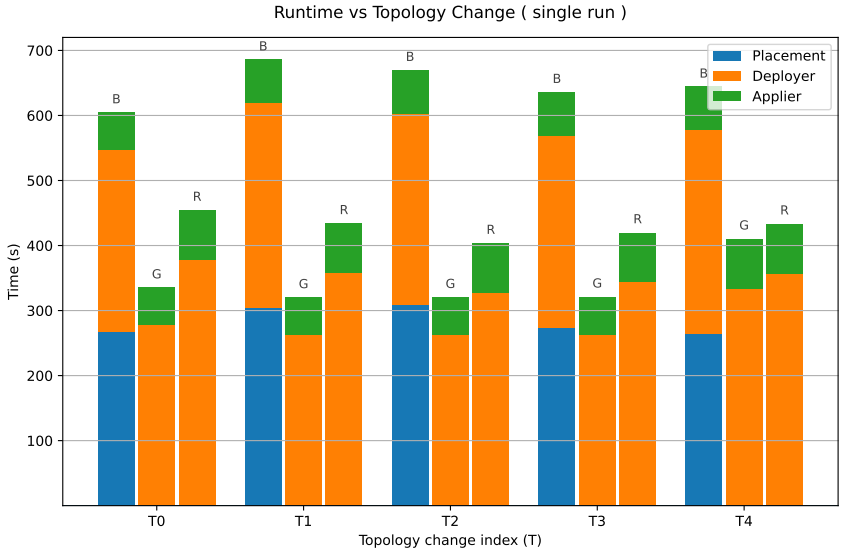

Figure 7.2: End-to-end execution time decomposition per topology change iteration

In the single-run scenario, the Brute Force algorithm exhibits the highest total execution time across all topology changes. This is mainly due to the larger placement computation phase, which reflects the combinatorial cost of exhaustive search. Although the deployment and configuration phases remain comparable to those of the other algorithms, the heavier computational effort results in systematically higher reaction times.

The Greedy strategy achieves the lowest total execution time in most topology changes. Its placement phase is significantly shorter than that of Brute Force, and this reduction directly translates into faster overall adaptation. Since deployment time remains similar across algorithms, the lower computational complexity of the greedy decision process provides a clear temporal advantage.

The Random algorithm occupies an intermediate position. Its placement computation is lightweight, sometimes comparable to Greedy. However, small variations in the generated configurations can slightly affect the configuration phase, producing moderate fluctuations in total execution time across topology indices.

To provide statistically robust insights, execution times were aggregated over 15 independent runs and represented through box plots (Figure 7.3). The multi-run distribution confirms and generalizes the trends observed in the single-run experiment.

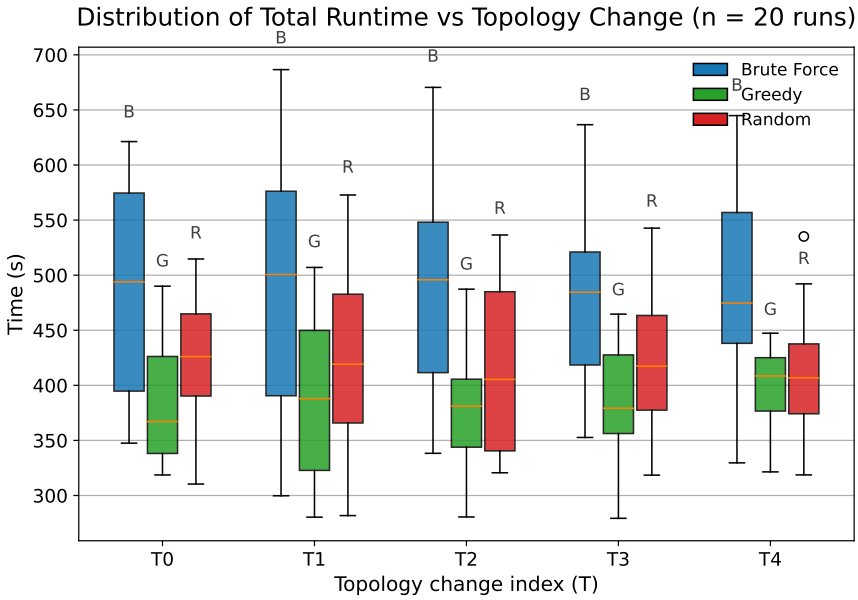

Figure 7.3: End-to-end execution time distribution over 20 runs

Across all topology changes, the Brute Force algorithm consistently shows the highest median execution time and the largest variability. Maximum values approach 700 seconds in some runs, highlighting the sensitivity of exhaustive search to topology conditions.

The Greedy strategy systematically achieves the lowest median reaction time. Compared to Brute Force, the median execution time is reduced by approximately 15–25%, with peaks approaching 30% depending on the topology index. Its interquartile ranges are generally narrower, indicating more predictable temporal behavior across runs.

The Random algorithm again lies between the two deterministic approaches. Its median execution time is consistently higher than Greedy but lower than Brute Force, with moderate dispersion reflecting the stochastic nature of placement

decisions.

Overall, the combined single-run and multi-run analyses indicate that algorithmic complexity primarily affects the placement computation phase, while infrastructure deployment remains the dominant contributor to total reaction time. Although Brute Force guarantees optimal placements, it incurs significantly higher and more variable temporal costs. Greedy provides the best trade-off between computational efficiency and temporal stability, whereas Random represents a compromise between simplicity and predictability. These results underline the importance of balancing optimality and reaction time in dynamic computing continuum environments.

7.4.3 Topological Stability Analysis

Topological stability was evaluated by measuring the Jaccard distance between consecutive PDC placement configurations under successive topology changes. This metric quantifies the structural variation of the deployed PDC set, with higher values indicating stronger reconfiguration.

Figure 7.4 illustrates the evolution of the Jaccard distance within a single experimental run. The results highlight how each topology modification impacts the placement stability of the three algorithms.

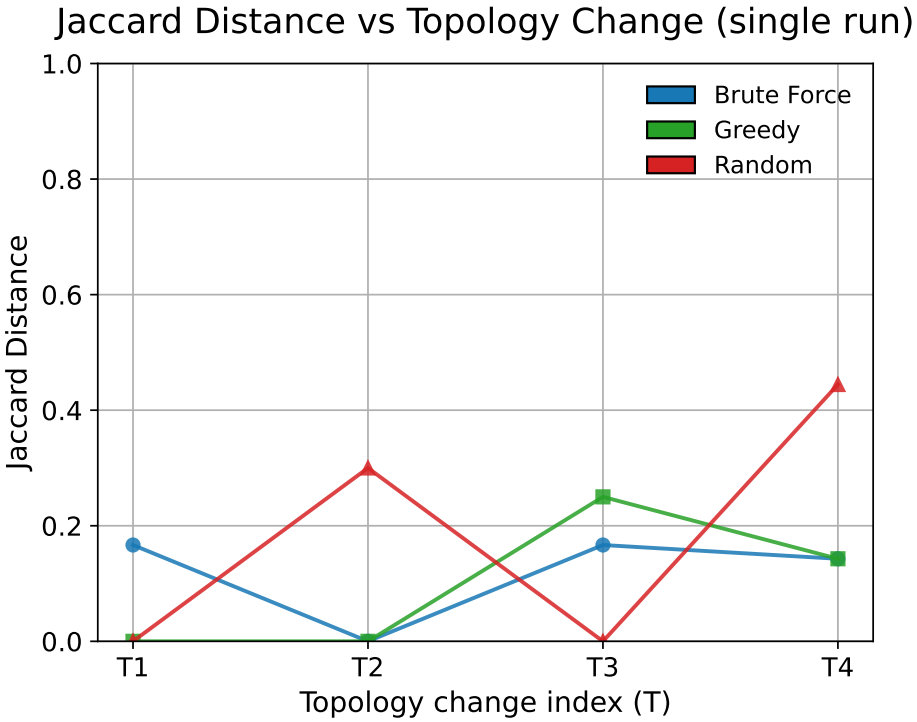

Figure 7.4: Jaccard distance evolution across topology changes.

In the single-run scenario, the Greedy strategy exhibits stable behavior during the initial topology changes. For T1 and T2, the Jaccard distance is equal to zero, indicating that the deployed PDC set remains unchanged despite network perturbations. At T3, the distance increases to approximately 0.25, suggesting a structural adjustment, while at T4 it decreases again (around 0.15), reflecting a limited reconfiguration.

The Brute Force algorithm shows moderate variability. At T2, the Jaccard distance is zero, meaning that the optimal configuration remains identical. For T1 and T3, the distance is around 0.17, indicating partial redeployment of PDC nodes. At T4, the value slightly decreases (approximately 0.14), confirming that topology changes may shift the global optimum, but typically without drastic structural disruption.

The Random strategy displays the highest instability, with significant oscillations across topology changes and a peak close to 0.45 at T4, confirming the stochastic nature of its placement decisions.

To obtain statistically robust insights, the Jaccard distance was aggregated over 15 independent runs and represented using box plots (Figure 7.5). The multi-run distribution confirms and generalizes the trends observed in the single-run analysis.

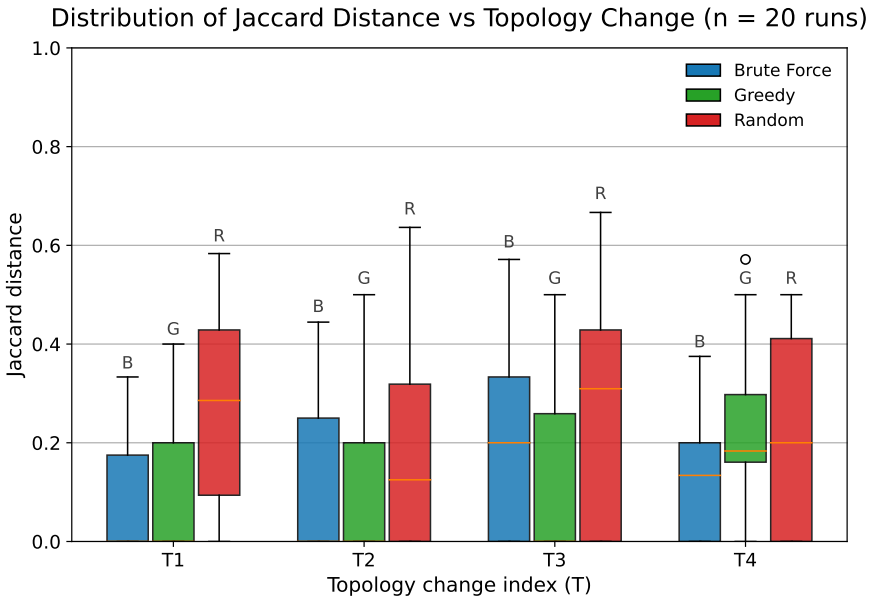

Figure 7.5: Jaccard distance distribution over 20 runs

Across most topology indices, the Greedy algorithm exhibits the lowest median Jaccard distance. Its median values are generally below those of Brute Force,

particularly at T2 and T3, indicating that the greedy strategy tends to preserve previously selected PDC nodes more effectively. Although some dispersion is present, especially at T1 and T4, the overall distribution remains comparatively compact.

The Brute Force algorithm shows slightly higher median values and comparable interquartile ranges. While it guarantees optimality at each iteration, small topology changes may shift the global optimum, resulting in partial but systematic reconfiguration. This explains the marginally higher structural churn compared to Greedy.

In contrast, the Random algorithm presents the largest variability and the highest maximum values across all topology changes. In particular, at T3 the upper whisker exceeds 0.6, highlighting substantial reconfiguration in several runs. Even when median values remain moderate, the dispersion confirms the intrinsic instability of stochastic placement.

Overall, the results indicate that the Greedy strategy provides the best trade-off between adaptability and structural continuity. Brute Force ensures optimal latency but introduces slightly higher reconfiguration due to optimal-solution shifts, whereas Random leads to significantly higher instability, making it less suitable for dynamic environments where minimizing infrastructure churn is critical.

7.4.4 Scalability Analysis

The scalability analysis focuses exclusively on the placement computation phase, isolating it from infrastructure provisioning and configuration overhead. This choice allows the evaluation of the intrinsic computational cost of the placement algorithms without being influenced by external factors such as container deployment time, network initialization, or database configuration.

Unlike the dynamic experiments presented in the previous section, no topological changes are introduced within a single run. Instead, the size of the network graph is progressively increased across successive runs by adding candidate nodes and PMUs, while keeping latency and bandwidth parameters constant. The Control Center node remains fixed, and the connectivity structure of the graph is preserved as the topology grows.

This controlled configuration allows the evaluation of how the intrinsic computational behavior of each placement strategy evolves as the problem size grows, independently from deployment-related costs. In particular, it highlights the scalability characteristics of the algorithms when applied to increasingly large monitoring infrastructures.

Figure 7.6 shows the placement computation time as a function of the total number of nodes in the graph (CC + candidate nodes + PMUs). The y-axis is represented on a logarithmic scale in order to clearly highlight differences in growth

trends and make exponential behaviors immediately visible. This representation allows a clearer comparison between algorithms whose execution time may differ by several orders of magnitude.

The results show three clearly differentiated scalability profiles.

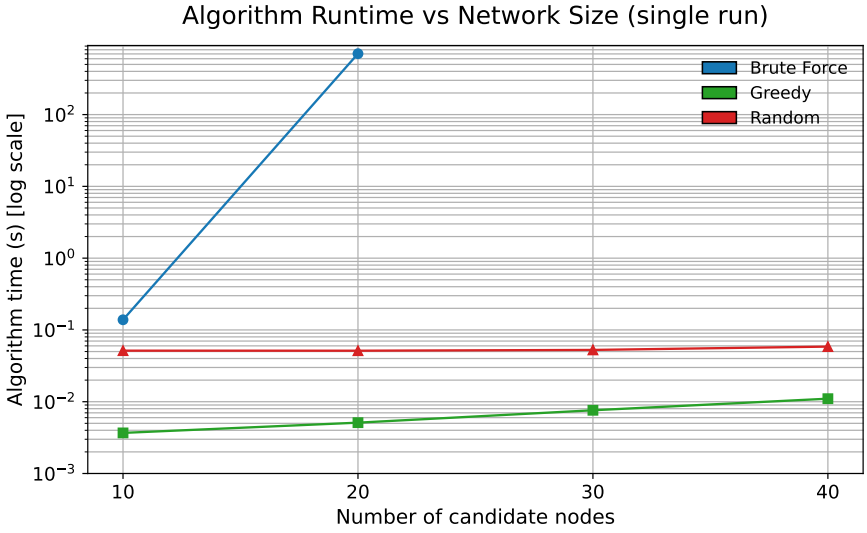

Figure 7.6: Placement computation time as a function of the number of nodes (logarithmic scale).

The brute-force strategy exhibits exponential growth. For small graphs, the execution time remains acceptable; however, as the number of nodes increases, the computation time rises sharply, reaching several minutes for mid-sized topologies. This behavior reflects the combinatorial explosion inherent in exhaustive subset exploration. As the number of candidate nodes increases, the number of possible PDC combinations grows exponentially, making the approach rapidly impractical for larger infrastructures.

In contrast, the Greedy strategy demonstrates an approximately linear growth trend. Even as the graph size increases significantly, the execution time remains in the order of milliseconds. This confirms that the heuristic selection mechanism effectively limits the expansion of the search space by focusing only on locally optimal feasible paths. Although this approach does not guarantee global optimality, it provides strong scalability properties, making it suitable for large-scale deployments.

The Random strategy appears almost constant across graph sizes in the single-run experiment. This behavior is primarily due to the bounded sampling mechanism adopted in its implementation. Since the number of random attempts and sampled paths per PMU is fixed, the computational effort does not grow proportionally

with the theoretical size of the solution space. Instead, it remains constrained by predefined exploration limits. However, this apparent constancy should not be interpreted as strict independence from problem size, but rather as a consequence of bounded stochastic exploration.

To better characterize variability and assess robustness, multiple independent runs were performed for each graph size. Figure 7.7 reports the distribution of placement computation times using box plots, again represented on a logarithmic scale.

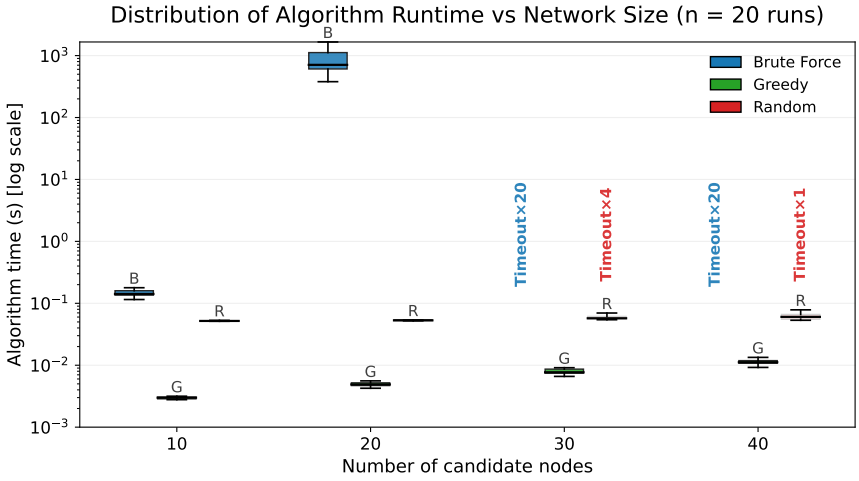

Figure 7.7: Distribution of placement computation time over 20 runs (logarithmic scale). Timeout occurrences are indicated above the corresponding configurations.

The multi-run results reinforce the limitations of the brute-force approach. For larger graph sizes, several executions exceed the imposed timeout threshold, confirming that exhaustive search does not scale reliably as the number of candidate nodes increases. Moreover, even when successful, execution times exhibit significant dispersion, highlighting the sensitivity of the combinatorial search to small structural variations in the graph.

The Greedy algorithm, instead, maintains consistently low execution times with minimal interquartile dispersion. This stability indicates that the heuristic decisions are largely deterministic and only marginally influenced by minor structural variations, resulting in predictable computational behavior across runs.

The Random strategy exhibits moderate variability. Although it generally remains significantly faster than Brute Force, at least one run exceeded the timeout limit for larger graph sizes. This confirms that the stochastic exploration process may occasionally require additional attempts before identifying a feasible solution,

depending on the specific order in which paths are sampled and on the constraints encountered during exploration. Therefore, while bounded in principle, the randomized strategy may still display occasional worst-case behavior under unfavorable sampling conditions.

Finally, scalability was also evaluated from a structural perspective. Figure 7.8 reports the number of selected PDCs (including the CC) as the number of candidate nodes increases.

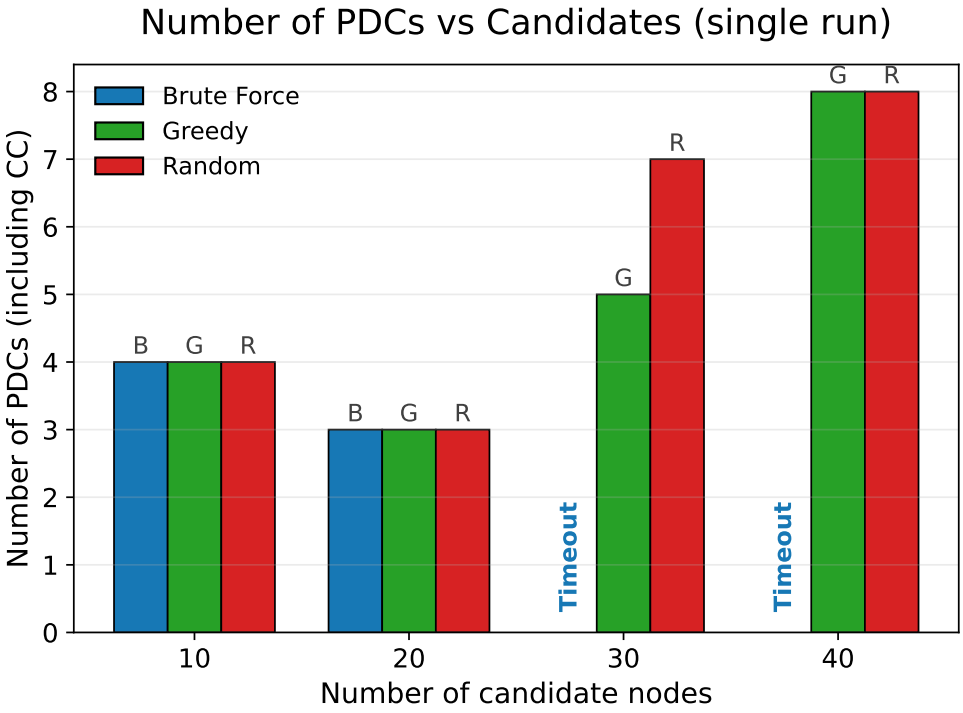

Figure 7.8: Number of selected PDCs (including the CC) as the number of candidate nodes increases.

For smaller graphs, all algorithms select the same number of PDCs, indicating that the solution space remains sufficiently constrained to allow convergence toward similar configurations. As the number of candidate nodes grows, differences become more evident. The Brute Force algorithm fails to complete within the timeout threshold for larger configurations, while Greedy and Random continue to produce feasible placements.

Between the two heuristic strategies, Greedy generally maintains a more controlled and gradual increase in the number of selected PDCs, producing relatively compact and structured configurations. The Random approach, instead, may select a slightly higher number of PDCs, reflecting its exploratory nature and reduced

structural guidance.

To further assess robustness and structural variability, the distribution of the number of selected PDCs was computed over 20 independent runs for each network size. Figure 7.9 reports the corresponding distribution, highlighting variability across runs and the presence of timeout events for larger configurations.

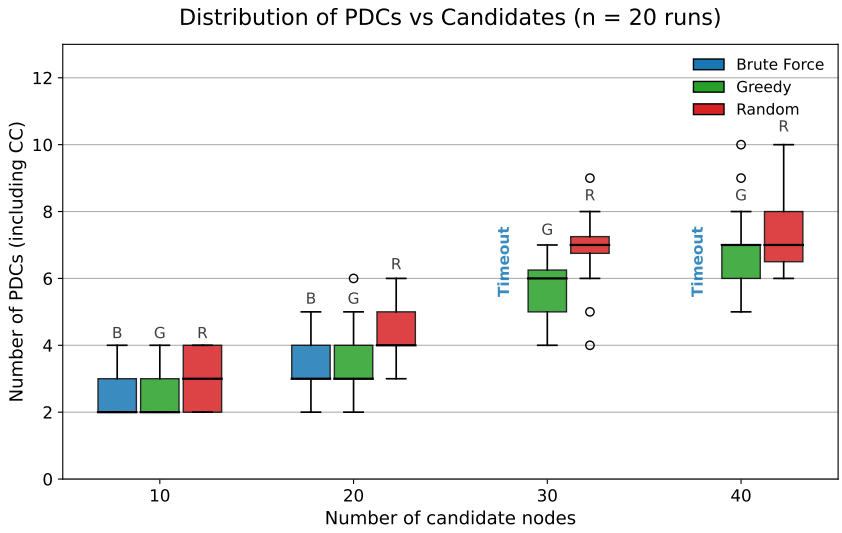

Figure 7.9: Distribution of the number of selected PDCs (including the CC) over 20 runs as the number of candidate nodes increases.

The boxplot representation confirms that, for small network sizes, all algorithms exhibit negligible variance and converge toward identical configurations. As the graph size increases, variability becomes more pronounced, particularly for the Random strategy. Greedy maintains a relatively compact interquartile range, indicating greater structural consistency across runs. Timeout occurrences for Brute Force in larger instances further confirm its limited scalability from a combinatorial perspective.

Overall, the scalability analysis highlights a fundamental trade-off between optimality, predictability, and computational feasibility. While Brute Force provides optimal solutions at small scale, it rapidly becomes impractical as the problem dimension increases. Greedy offers the most balanced behavior, combining strong scalability, low variability, and structural efficiency. Random provides acceptable computational performance but introduces higher variability and less predictable structural outcomes, particularly for larger topologies.

7.5 Discussion and Overall Assessment

The experimental evaluation presented in this chapter provides a comparative assessment of the three placement strategies along three main dimensions: temporal performance, topological stability, and scalability.

The results show that algorithmic optimality alone is not sufficient to determine practical suitability in dynamic and distributed environments. Placement strategies must instead be evaluated by jointly considering computational efficiency, structural continuity, and operational overhead.

The main findings are summarized in Table 7.1, which consolidates the behavior of each algorithm across the considered metrics.

Table 7.1: Comparative Summary of Placement Algorithms

Algorithm	**Latency Optimality**	**Reaction Time**	**Topological Stability**	**Scalability Behavior**
Brute Force	Global optimum	High and variable	Moderate	Exponential growth, timeouts
Greedy	Near-optimal	Low and stable	High	Near-linear growth
Random	Feasible but stochastic	Moderate	Low (high variability)	Bounded but variable, occasional timeout

Brute Force guarantees globally optimal latency whenever a feasible solution exists, but its exponential computational growth and high temporal variability strongly limit its applicability in large-scale or highly dynamic infrastructures.

Greedy emerges as the most balanced approach. It achieves near-optimal placements, maintains low and predictable reaction times, and exhibits strong structural stability under topology changes. Its near-linear scalability further confirms its suitability for increasing network sizes.

Random benefits from bounded computational effort and delivers acceptable performance in many scenarios. However, its stochastic exploration introduces higher variability in both execution time and structural stability, including occasional timeout events in larger topologies, thus reducing predictability.

Overall, the results indicate that heuristic approaches are the most practical solution for latency-aware PDC placement in computing continuum environments. Among them, Greedy offers the best compromise between optimality, stability, and scalability, making it the most suitable candidate for real-world deployment.

Chapter 8

Future Work

Although this thesis demonstrates the feasibility of latency-aware PDC placement in a multi-cluster Kubernetes environment, several research directions remain open for further investigation and improvement. Future developments can be grouped into three main areas: optimization robustness and stability, architectural evolution within the computing continuum, and the integration of intelligent learning-based strategies.

8.1 Optimization and Stability Improvements

The first line of future research concerns the evolution of the placement formulation itself. While the current work focuses primarily on feasibility under latency and bandwidth constraints, practical deployments in dynamic environments require a stronger emphasis on stability and maintenance-aware optimization. In computing continuum scenarios, minimizing performance metrics alone is often insufficient; structural consistency and operational continuity become equally critical.

8.1.1 Maintenance-Aware PDC Placement

In the current formulation, the primary optimization objectives focus on latency constraints and bandwidth feasibility. However, in practical deployments, the number of modifications applied to the infrastructure between successive placement decisions is equally relevant.

While an initial deployment may tolerate the installation of a higher number of PDC instances to guarantee feasibility, subsequent reconfigurations should ideally avoid excessive structural changes. Frequent addition or removal of PDCs increases operational complexity, deployment overhead, and potential service disruption.

A promising research direction therefore consists in explicitly incorporating

the number of PDC modifications as an optimization parameter. This could be modeled in different ways:

- minimizing the absolute number of PDC nodes deployed,
- minimizing the number of newly added PDCs with respect to the previous configuration,
- minimizing the total churn (additions + removals) across iterations,
- introducing weighted penalties for structural changes during maintenance phases.

Such an approach would enable the differentiation between two operational modes:

- an *initial deployment phase*, where feasibility and latency minimization dominate,
- a *maintenance phase*, where stability and minimal structural variation become priority objectives.

From an optimization perspective, this extension transforms the placement problem into a multi-objective or state-dependent optimization problem, where the previous configuration becomes part of the decision state. This perspective aligns naturally with computing continuum environments, where infrastructure evolution must balance performance optimization and operational stability.

8.1.2 Sensitivity Analysis on Latency Constraints

A complementary research direction concerns the systematic analysis of the maximum end-to-end latency constraint.

In the current experiments, the latency threshold is treated as a fixed parameter. However, varying this constraint across a range of values could provide deeper insight into the structural behavior of the placement algorithms.

In particular, a parametric sensitivity study could reveal:

- the trade-off between the number of deployed PDCs and achievable latency,
- the stability of placement decisions under tighter or looser constraints,
- the scalability limits of each algorithm when stricter QoS requirements are imposed.

Such an analysis would allow the identification of critical threshold values beyond which certain algorithms become infeasible or excessively costly. This would support the design of adaptive QoS-aware placement policies in dynamic environments.

8.2 Infrastructure Evolution within the Computing Continuum

Beyond optimization refinements, a second macro-area of future work concerns the architectural and infrastructural evolution of the system. The current framework already operates within a distributed multi-cluster Kubernetes environment, but further extensions could strengthen its alignment with computing continuum principles, particularly in terms of real-time adaptivity and hierarchical flexibility.

8.2.1 Real-Time Network Monitoring Integration

The current implementation assumes that network parameters such as latency, bandwidth, and link availability are known and updated externally before each placement recomputation. A natural extension would consist in integrating real-time network telemetry mechanisms directly into the orchestration loop.

This could be achieved through multiple approaches, including:

- active probing techniques (e.g., periodic latency measurements between clusters),
- passive monitoring through Kubernetes metrics and CNI plugins,
- integration with observability stacks such as Prometheus and service mesh telemetry.

By continuously collecting real-time network metrics, the placement framework could evolve toward a fully closed-loop adaptive control system, capable of triggering reconfiguration decisions automatically based on measured performance degradation rather than predefined topology changes.

8.2.2 Hierarchical Multi-Output Extensions in openPDC

The current openPDC architecture assumes a strictly hierarchical aggregation model, where each PDC forwards its output toward a single higher-level concentrator.

An interesting architectural extension would be to enable controlled output splitting, allowing a PDC to forward synchronized data streams toward multiple higher-level PDCs simultaneously. Such a modification could:

- improve bandwidth utilization through traffic distribution,
- increase resilience by introducing redundant aggregation paths,
- enable load balancing across multiple Control Centers.

Implementing this feature would require modifications to the openPDC configuration model and possibly to its internal stream management logic. From an optimization perspective, this would also transform the placement problem into a multi-path flow allocation problem, significantly increasing its complexity and opening new research challenges.

8.3 Intelligent and Learning-Based Placement Strategies

The final direction of future work concerns the integration of intelligent decision-making mechanisms capable of learning adaptive placement policies directly from network behavior.

8.3.1 Reinforcement Learning and Graph-Based Approaches

While this thesis evaluates deterministic and heuristic algorithms, future research could explore reinforcement learning or graph neural network-based approaches capable of learning adaptive placement strategies.

Such methods could:

- learn policies that balance latency, bandwidth, and stability,
- reduce recomputation overhead through incremental updates,
- generalize to unseen topologies,
- incorporate historical placement data into the decision process.

Given the graph-structured nature of the problem and the state-dependent dynamics introduced by maintenance-aware optimization, learning-based models represent a promising direction for large-scale computing continuum environments.

Chapter 9

Conclusions

This thesis addressed the problem of latency-aware Phasor Data Concentrator (PDC) placement within distributed, multi-cluster Kubernetes environments, with the objective of bridging theoretical placement optimization and practical infrastructure orchestration in computing continuum scenarios.

The work began by formalizing the placement problem as a constrained graph optimization task. The network was modeled as a graph with latency, bandwidth, and availability constraints, and the placement of PDC nodes was defined as the search for feasible PMU–PDC–Control Center chains satisfying end-to-end delay and resource limitations. This abstraction provided a *flexible and general modeling foundation*, enabling the evaluation of different placement strategies under controlled experimental conditions.

Three algorithmic approaches were implemented and compared: a Brute Force method, a Dijkstra-based Greedy strategy, and a Random exploratory algorithm. Each of these approaches represents a different trade-off between optimality, computational complexity, and adaptability. The Brute Force algorithm guarantees optimal solutions but exhibits exponential growth in computational cost; the Greedy approach offers fast and stable reconfiguration with limited global awareness; the Random strategy provides exploratory diversity but limited efficiency and predictability.

A key contribution of this thesis lies in the integration of placement logic with real infrastructure deployment. Beyond algorithmic modeling, a fully *automated pipeline* was designed to transform abstract placement outputs into concrete multi-cluster Kubernetes deployments. Through the combination of placement computation, cluster provisioning, database initialization, and automated configuration replay, the system enables deterministic regeneration of the infrastructure following topology changes. This integration demonstrates that latency-aware placement decisions can be operationalized in realistic environments rather than remaining purely theoretical constructs.

The experimental evaluation provided insights into three complementary dimensions: temporal performance, topological stability, and scalability. Results show that infrastructure provisioning and configuration overhead often dominate pure placement computation time, highlighting the importance of evaluating end-to-end orchestration cost. The Greedy strategy consistently demonstrated the lowest reaction time under dynamic conditions, while the Brute Force method served as a benchmark for optimality at the cost of scalability. Topological stability analysis, based on Jaccard distance, revealed varying degrees of structural churn across algorithms, suggesting that placement quality must be evaluated not only in terms of latency but also in terms of infrastructure disruption.

Overall, this work demonstrates that latency-aware PDC placement can be effectively embedded into a multi-cluster Kubernetes ecosystem, forming a closed-loop system capable of adapting to network changes while maintaining reproducibility and operational consistency. By unifying graph-based optimization, automated configuration, and container orchestration, the proposed framework contributes toward practical realization of adaptive monitoring infrastructures in *computing continuum environments*.

The results obtained provide both a validated implementation framework and a foundation for future research directions, including maintenance-aware optimization, real-time telemetry integration, architectural extensions of openPDC, and learning-based placement strategies.

In conclusion, this thesis demonstrates that the integration of graph-based optimization models, automated deployment pipelines, and cloud-native orchestration mechanisms can effectively support latency-sensitive monitoring services across distributed infrastructures.

By embedding placement decisions within an *intent-driven orchestration* workflow, the proposed framework moves beyond static deployment strategies and enables adaptive infrastructure reconfiguration aligned with performance objectives and operational constraints.

This convergence between optimization logic and intent-based orchestration represents a concrete step toward realizing computing continuum architectures in which network conditions, resource availability, and application requirements are continuously reconciled through automated control loops.

Appendix A

Automation pipeline

Algorithm: autopdc_configurator

Create initial graph G and draw it
repeat
    Update G:
        modify_latency(G)
        modify_edge_status(G)
        modify_bandwidth(G)
    $alg \leftarrow$ CHOOSE_ALGORITHM()
    $(pdcs, paths, L_{\max}) \leftarrow$ RUN(alg, G)
    DRAW(G, $pdcs$, $paths$)
    Normalize node names in $paths$
    Write `output.json` with relevant placement data
    if `SKIP_DEPLOY` = false **then**
        Run `deployer.sh` (log runtime)
        if deployment succeeded AND SKIP_DELAY = false **then**
            Run `applier.py` (log runtime)
        end if
    end if
    Log total iteration time
until user chooses to stop

Algorithm: deployer

$clusters \leftarrow$ READ(`output.json`)
for all $c \in clusters$ **do**
    if c not in `k3d cluster list` **then**
        Create cluster c with required port mappings
    end if
end for
if cluster database exists **then**
    Clear cluster database

```
    Delete existing PDC instances
end if
Merge kubeconfig files
Apply Percona.yaml
for all c ∈ clusters do
    Modify openpdc.yaml according to cluster c
    Apply openpdc.yaml using kubectl
end for
for all PMU in configuration do
    Apply pmu.yaml using kubectl
end for
```

Algorithm: applyer

```
data ← read(output.json)
config ← pdc_topology(data)
order ← compute_order(data)
execute(config, order)
```

Algorithm: pdc_topology

```
Initialize config ← ∅
for all path ∈ data.paths do
    pmu ← first element of path
    clusters ← remaining elements of path
    for all c ∈ clusters do
        if c ∉ config then
            Initialize config[c] with:
                pmu_direct ← ∅
                outputstream ← ∅
                connections_downstream ← ∅
        end if
    end for
    leaf ← clusters[1]
    Add pmu to config[leaf].pmu_direct
    Add pmu to config[leaf].outputstream
    for i = 2 to |clusters| do
        child ← clusters[i − 1]
        parent ← clusters[i]
        Add pmu to config[parent].outputstream
        if downstream connection (parent → child) exists then
            Add pmu to its PMU list
        else
            Create new downstream connection:
                node ← child
                pmus ← [pmu]
                port ← calc_port(child)
```

```
                Append connection to config[parent].connections_downstream
            end if
        end for
    end for
    for all c ∈ config do
        Sort config[c].outputstream by PMU index
    end for
    return config
```

Algorithm: compute_order

```
Initialize deps ← ∅, all_clusters ← ∅
for all path ∈ data.paths do
    Extract cluster sequence (exclude PMU)
    Add clusters to all_clusters
    for all consecutive pairs (child, parent) do
        Add child as dependency of parent in deps
    end for
end for
for all c ∈ all_clusters do
    if c ∉ deps then
        deps[c] ← ∅
    end if
end for
Initialize in_degree[c] ← 0 for all clusters
for all parent ∈ deps do
    for all child ∈ deps[parent] do
        Increment in_degree[parent]
    end for
end for
queue ← {c | in_degree[c] = 0}
order ← ∅
while queue ≠ ∅ do
    node ← pop first element from queue
    Append node to order
    for all parent depending on node do
        Decrement in_degree[parent]
        if in_degree[parent] = 0 then
            Add parent to queue
        end if
    end for
end while
return order
```

Algorithm: execute

```
for all cluster ∈ order do
```

2: Run `createhistorian`
3: Run `addinput`
4: Run `connectiontopdc`
5: Run `createoutputstream`
6: **end for**

Bibliography

[1] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. «Edge computing: Vision and challenges». In: *IEEE internet of things journal* 3.5 (2016), pp. 637–646 (cit. on p. 7).

[2] Pedro Garcia Lopez, Alberto Montresor, Dick Epema, Anwitaman Datta, Teruo Higashino, Adriana Iamnitchi, Marinho Barcellos, Pascal Felber, and Etienne Riviere. *Edge-centric computing: Vision and challenges*. 2015 (cit. on p. 7).

[3] Luciano Baresi, Danilo Filgueira Mendonça, Martin Garriga, Sam Guinea, and Giovanni Quattrocchi. «A unified model for the mobile-edge-cloud continuum». In: *ACM Transactions on Internet Technology (TOIT)* 19.2 (2019), pp. 1–21 (cit. on p. 7).

[4] Orazio Tomarchio, Domenico Calcaterra, and Giuseppe Di Modica. «Cloud resource orchestration in the multi-cloud landscape: a systematic review of existing frameworks». In: *Journal of Cloud Computing* 9.1 (2020), p. 49 (cit. on p. 7).

[5] Intel Corporation. *Platform Aware Scheduling*. `https://github.com/intel/platform-aware-scheduling`. Accessed: Jan. 2026 (cit. on p. 7).

[6] Angelo Marchese and Orazio Tomarchio. «Network-aware container placement in cloud-edge kubernetes clusters». In: *2022 22nd IEEE international symposium on cluster, cloud and internet computing (CCGrid)*. IEEE. 2022, pp. 859–865 (cit. on p. 7).

[7] Estela Carmona-Cejudo, Francesco Iadanza, and Muhammad Shuaib Siddiqui. «Optimal offloading of Kubernetes pods in three-tier networks». In: *2022 IEEE Wireless Communications and Networking Conference (WCNC)*. IEEE. 2022, pp. 280–285 (cit. on p. 7).

[8] Qian Li, Bin Li, Pietro Mercati, Ramesh Illikkal, Charlie Tai, Michael Kishinevsky, and Christos Kozyrakis. «RAMBO: Resource allocation for microservices using Bayesian optimization». In: *IEEE Computer Architecture Letters* 20.1 (2021), pp. 46–49 (cit. on p. 8).

[9] Amit Samanta, Yong Li, and Flavio Esposito. «Battle of microservices: Towards latency-optimal heuristic scheduling for edge computing». In: *2019 IEEE Conference on Network Softwarization (NetSoft)*. IEEE. 2019, pp. 223–227 (cit. on p. 8).

[10] Samodha Pallewatta, Vassilis Kostakos, and Rajkumar Buyya. «Microservices-based IoT application placement within heterogeneous and resource constrained fog computing environments». In: *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing*. 2019, pp. 71–81 (cit. on p. 8).

[11] Ram Govinda Aryal and Jörn Altmann. «Dynamic application deployment in federations of clouds and edge resources using a multiobjective optimization AI algorithm». In: *2018 Third international conference on fog and mobile edge computing (FMEC)*. IEEE. 2018, pp. 147–154 (cit. on p. 8).

[12] Claudio Usai. «Delivering Resilient Virtualized Services in Smart Grid Environments». PhD thesis. Politecnico di Torino, 2021 (cit. on p. 8).

[13] Riccardo Medina. «Edge-to-Cloud Multi-Cluster Orchestration for Smart Grid Monitoring Services». PhD thesis. Politecnico di Torino, 2024 (cit. on p. 8).

[14] Claudio Lorina. «Resilient and Low Latency Communications in Smart Grid Environments». PhD thesis. Politecnico di Torino, 2021 (cit. on p. 8).

[15] Radoslav Furnadzhiev. «An Experimental Evaluation of Latency-Aware Scheduling for Distributed Kubernetes Clusters». In: *Engineering Proceedings* 100.1 (2025), p. 25 (cit. on p. 9).

[16] Cristopher Chiaro, Doriana Monaco, Alessio Sacco, Claudio Casetti, and Guido Marchetto. «Latency-aware scheduling in the cloud-edge continuum». In: *NOMS 2024-2024 IEEE Network Operations and Management Symposium*. IEEE. 2024, pp. 1–5 (cit. on p. 9).

Acknowledgements

Ed eccoci qua, alla parte dei ringraziamenti. Partirei dalla mia famiglia: grazie a mia sorella Elena, che mi ha insegnato a buttarmi, a fare cose nuove, anche se poi magari si sbatte la testa. A mia mamma, che mi ha insegnato la costanza e ad avere disciplina nelle cose che andiamo a fare, e a mio papà, che mi ha insegnato che lo studio non è tutto, che c'è molto altro da fare e che, ci ho messo un po' a capirlo lo ammetto, alcune volte staccare un po' fa bene.

Ringrazio poi i nonni, Maria Teresa, Luciana, Francesco e Antonio. Il tempo passato con voi è stato, e lo è ancora, carburante per me, energia pura, con voi mi sento a casa ovunque io sia. E ringrazio anche te Man, è stato bello essere coinquilini, il tempo è volato.

Infine, ringrazio Andrea, compagna di avventure. Potrei dire la solita cosa "grazie di avermi supportato e sopportato ecc", e invece dirò che ti ho sopportato io. A parte gli scherzi, ci siamo conosciuti che avevo una manciata di crediti in triennale, ma con te è stato tutto più facile. Tutto quello che mi metteva ansia, la tua simpatia, vicinanza e i tuoi modi, me l'hanno reso più chiaro e mi hanno concentrato e caricato ancora di più. Grazie pati

E ora veniamo a voi amici. Premetto che non posso mettere tutti. Quindi, comincio a ringraziare te che stai leggendo queste parole, e che quindi probabilmente sei e sei stato parte in qualche modo parte del mio percorso. In particolare poi, ringrazio Tony e Carlo: la nostra amicizia non si misura in quanto ci vediamo, ma nell'energia di quando lo facciamo. Nel momento del bisogno ci siete sempre stati. Grazie a Teo e Riki, compagni di scuola ma anche al di fuori, ne abbiamo fatte tante. E grazie ai miei compagni di uni: Mattia, Loris, Zeno e Alessia. Mi ricordo benissimo la prima volta che ci siamo visti: al lab di architetture, e non avevamo la più pallida idea di come fare quell'esercizio. Ma alla fine, ce l'abbiamo fatta tutti. Avete reso questo percorso più speciale.

Ringrazio inoltre il relatore Prof. Risso, il co-relatore interno Stefano e i co-relatori esterni Elisa e Luca per la loro disponibilità, umanità e professionalità dimostrate durante lo sviluppo di questo lavoro.

Infine, voglio ringraziare me. Voglio dire grazie Dario per non aver mollato, e per essere andato fino in fondo. In realtà, non ho mai dubitato un secondo che questa fosse la strada giusta.