Energy Efficient Implementation of Data Exchange Archetypes in DYNAMOS

DYNAMOS: Dynamically Adaptive Microservice-based OS

Collin Poetoehena
collinpoetoehena@gmail.com

May 15, 2025, 109 pages

Academic supervisor: dr. Ana Oprescu, a.m.oprescu@uva.nl
Second reader: dr. Shashikant Ilager, s.s.ilager@uva.nl
Research group: UvA: Complex Cyber-Infrastructure (CCI), https://cci-research.nl/

Universiteit van Amsterdam
Faculteit der Natuurwetenschappen, Wiskunde en Informatica
Master Software Engineering
http://www.software-engineering-amsterdam.nl

Abstract

As global concerns over energy consumption and ecological sustainability continue to grow, optimizing energy efficiency in software systems has become an increasingly critical research area. The ICT sector contributes significantly to global greenhouse gas emissions, with cloud computing and microservice-based architectures playing a substantial role in this energy demand. To address this challenge, this thesis investigates energy efficiency optimizations for data exchange archetypes within DYNAMOS, a dynamically adaptive microservice-based middleware.

The research first establishes a structured energy efficiency report pipeline to identify root causes of energy inefficiencies in DYNAMOS. Based on this analysis and a supporting literature review, nine candidate optimizations are identified. Two of these—caching requests and transferred data compression—are implemented and empirically evaluated to assess their effect on energy consumption and execution time. A key strength of this thesis lies in its validation across two different environments: a local Docker Desktop setup and a distributed Kubernetes cluster. The distributed setup was deployed on the FABRIC testbed to ensure broader applicability of the findings.

Experimental results show that caching consistently provides significant improvements in both energy consumption and performance. In contrast, transferred data compression generally shows no significant benefit and can even introduce computational overhead, negatively affecting performance. Additional findings reveal a moderate correlation between execution time and energy consumption, which was stronger in the local environment and weaker in the distributed FABRIC setup—underscoring the complexity of energy behavior in real-world deployments. Finally, this study identifies challenges related to software-based energy estimation, container selection strategies, and the influence of architectural decisions on measurement precision and system stability.

This work contributes to energy efficiency by demonstrating how targeted optimizations and energy-aware monitoring can enhance the sustainability of microservice-based systems. It further offers methodological guidance for future research into energy-efficient Software Engineering. Future work may expand on these findings by evaluating additional optimizations, quantifying the energy overhead of monitoring infrastructure, and refining energy measurement methodologies in virtualized environments to enhance the accuracy of reported energy metrics.

Contents

Chapter 1

Introduction

The importance of energy efficiency in Information Communication Technology (ICT) has heightened with the expansion of data centers and cloud computing [1]. Freitag *et al.* [2] suggest that the ICT sector's share of global greenhouse gas (GHG) emissions could be as high as 2.1%–3.9% and is projected to grow in the future. Therefore, addressing energy efficiency in ICT is not only a technical challenge but also a societal imperative.

Among various ICT architectures, microservice-based systems have gained significant traction due to their scalability and flexibility. However, these systems also introduce additional overhead and complexity that can impact energy consumption. Improving the energy efficiency of such systems is therefore a crucial step toward sustainable software [1–4].

Dynamically Adaptive Microservice-based Operating System (DYNAMOS) is a self-adaptive middleware system that abstracts complex data exchange patterns into standardized microservices. The system is designed to dynamically compose these microservices to meet various policy and user requirements. The 'operating system' within DYNAMOS allows the optimization of extra functional properties, enabling energy savings based on the environment of the system [5].

This thesis investigates the integration of energy efficiency capabilities into DYNAMOS, aiming to enhance its sustainability. Specifically, the main goal is to develop energy efficiency optimizations that allow archetypes to be implemented in a more energy-efficient way. First, an 'energy efficiency report pipeline' is implemented in DYNAMOS to identify root causes of inefficiencies. Second, targeted energy efficiency optimizations are researched and developed for integration into DYNAMOS' archetypes.

Through this approach, the thesis seeks to improve the energy efficiency of archetype implementations, thereby supporting global efforts toward sustainable development.

1.1 Problem Statement

The energy consumption of software systems has emerged as a critical concern in both academia and industry. Although awareness of software sustainability is increasing, research in this field remains underdeveloped [6]. Balanza-Martinez *et al.* [6] emphasize that algorithms and strategies for improving energy efficiency require deeper investigation, urging researchers and practitioners to prioritize this area to advance global sustainability goals. A key gap highlighted in the literature is the lack of empirical studies on energy efficiency optimizations, particularly the scarcity of actionable, implementable solutions.

DYNAMOS presents a promising platform for addressing this gap. As described by Stutterheim [5], DYNAMOS offers a framework and methodology for dynamically selecting and composing archetypes. While DYNAMOS already supports adaptation based on various policies, its potential to optimize energy consumption remains largely unexplored. Its ability to orchestrate archetypes dynamically creates new opportunities to integrate energy-efficient strategies directly into system behavior, making it an ideal testbed for evaluating energy efficiency optimizations.

This thesis addresses these gaps by identifying and assessing energy efficiency optimizations for data exchange archetypes, leveraging DYNAMOS to evaluate their effectiveness. To achieve this, several key challenges were tackled. First, an energy efficiency report pipeline was implemented within DYNAMOS to identify the root causes of energy consumption. Inspired by the work of Floroiu *et al.* [1], this pipeline begins with energy consumption measurements and employs an Anomaly Detection (AD) algorithm to detect deviations or abnormalities. A subsequent Root Cause Analysis (RCA) algorithm then determines the underlying causes of these deviations. Second, a comprehensive literature review was conducted

to identify relevant energy efficiency optimizations for data exchange archetypes. Finally, structured experiments were performed to empirically assess the real-world impact of the implemented optimizations.

1.1.1 Research Questions

To tackle these issues, we investigate the following research questions:
RQ1 What energy efficiency optimizations are suitable for data exchange archetypes?
RQ2 To what extent do these optimizations enhance the energy efficiency of data exchange archetypes?

1.1.2 Research Method

The research presented in this thesis is conducted using **Technical Action Research (TAR)** [7], an artifact-based research methodology tailored to develop, implement, and iteratively refine technical solutions within a real-world context. TAR is particularly suited for this study as it enables continuous learning through cycles of artifact creation, deployment, and evaluation, while accommodating adjustments based on empirical findings [7, 8]. Given the time constraints of a master's thesis project, the TAR cycles were deliberately limited to two primary iterations: one in a local environment and one in the FABRIC environment.

The research process began with the establishment of a baseline, where the existing energy consumption of DYNAMOS was analyzed in detail. This initial phase involved developing and deploying an energy efficiency report pipeline based on state-of-the-art methods for energy monitoring and root cause analysis. Primary contributors to energy inefficiencies in DYNAMOS were systematically identified by leveraging tools such as Prometheus and Kepler and applying algorithms for anomaly detection and root cause discovery. In parallel, a literature review was conducted to explore known energy efficiency optimizations, resulting in the selection of candidate strategies tailored to the identified bottlenecks.

Building upon the baseline analysis, the first iteration implemented two selected energy efficiency optimizations: caching of request results and compression of transferred data. These optimizations were integrated into two DYNAMOS archetypes used within the 'Universities of the Netherlands' (UNL) use case. Their effectiveness was empirically validated through structured experiments conducted in a local environment, specifically using a single-node Kubernetes cluster deployed on Docker Desktop with WSL2 virtualization. The local experiments were designed to systematically measure the improvements in energy consumption and execution time achieved by the optimizations, while maintaining consistent experimental conditions to ensure reliability of results.

The second iteration extended the validation of the implemented optimizations to a distributed environment by deploying DYNAMOS on FABRIC, a large-scale programmable testbed for computer science research. A multi-node Kubernetes cluster was set up with services distributed across independent virtual machines to simulate realistic, distributed data exchange scenarios. The experiments on FABRIC replicated the local tests but under different infrastructural conditions, allowing assessment of the portability, scalability, and robustness of the optimizations across multiple environments.

Validation of the research was conducted through both theoretical and empirical approaches. The theoretical validation involved cross-referencing the identified optimizations with findings from the scientific literature to confirm their expected energy benefits. Empirical validation, on the other hand, was achieved by executing controlled experiments in both environments and applying statistical analysis to measure the energy savings and performance improvements. Particular attention was paid to challenges such as anomaly detection, variability due to virtualization artifacts, and differences in energy reporting granularity between the WSL2 and FABRIC setups.

All research activities were carried out within the University of Amsterdam (UvA) under the Complex Cyber-Infrastructure (CCI) research group[1], ensuring an academic environment with access to expertise in distributed systems and sustainable computing.

1.2 Contributions

This research makes several contributions in both academic and technical domains, advancing the understanding of energy efficiency optimizations in data exchange archetypes and microservice-based architectures.

[1] https://ivi.uva.nl/research/complex-cyber-infrastructure.html

1.2.1 Core Research Contributions

CR1 Systematic identification and analysis of energy efficiency optimizations relevant to data exchange archetypes. This includes a conceptual classification of optimizations and a practical evaluation framework tailored to data exchange systems, with a specific implementation focus on DYNAMOS (see Chapter 4).
CR2 Comprehensive empirical evaluation of caching and compression optimizations, analyzing their impact on energy consumption and performance across both local (single-node) and distributed (multi-node FABRIC) environments (see Chapter 7).
CR3 Development of a structured methodology for identifying and reporting the root causes of energy consumption in Kubernetes-based microservices through container-level monitoring, based on an adapted reporting pipeline (see Chapter 3).
CR4 Investigation of energy efficiency challenges in microservice-based architectures, highlighting trade-offs between execution time, concurrency, and orchestration overhead in distributed systems (see Chapter 8).
CR5 Evaluation of energy measurement methodologies in Kubernetes clusters, including techniques for accurate container selection, energy attribution, and Prometheus-based data aggregation (see Section 3.2).
CR6 Investigation of the reliability of energy monitoring in VM-based environments, with a specific focus on Docker Desktop under WSL2, revealing notable measurement discrepancies between the local and FABRIC environments (see Sections 3.1.3 and 8.1.3).
CR7 Deployment and use of a real-world distributed Kubernetes environment (FABRIC) to validate findings under scalable infrastructure conditions, demonstrating increased measurement granularity, system stability, and orchestration efficiency (see Section 5.3 and Chapter 7).
CR8 Empirical analysis of the correlation between execution time and energy consumption, showing environment-specific behavior (see Chapters 7 and 8).
CR9 Additional experimental exploration into system behavior under architectural variations, including Jaeger-based tracing analysis and controlled Kubernetes performance testing, which revealed nuanced infrastructure-level differences in DYNAMOS execution (see Section 7.6).

1.2.2 Core Technical Contributions

CT1 Integration of energy monitoring capabilities into DYNAMOS, enabling fine-grained energy consumption analysis (see Section 3.1).
CT2 Implementation of energy optimization strategies—specifically caching and data compression—tailored for data exchange archetypes within DYNAMOS, and demonstrating their effectiveness in both local single-node and distributed multi-node environments (see Chapters 5 and 7).
CT3 Adaptation and extension of an existing energy efficiency reporting pipeline [1] for use in both Kubernetes and VM-based environments. Special attention was given to ensure compatibility with WSL2 setups (see Section 3.1).
CT4 Open-source contributions to the DYNAMOS codebase, including bug fixes, energy efficiency enhancements, and the deployment in FABRIC. Contributions were published in a publicly available GitHub repository[2].
CT5 Complete deployment and orchestration of a multi-node Kubernetes cluster on the FABRIC testbed for running DYNAMOS. This includes the custom setup of Kubernetes using Kubeadm, the selection and troubleshooting of the Flannel Container Network Interface (CNI) plugin for pod networking, and a distributed architecture mapping DYNAMOS participants to separate VMs for accurate energy experimentation (see Section 5.3).
CT6 Development of a monitoring and remote access setup in FABRIC, including integration of Kepler with Grafana dashboards, exposure of services through secure SSH tunneling, and the configuration of NodePort services for Prometheus and Grafana (see Section 5.3).
CT7 Empirical validation of the portability of optimizations across environments. The caching and compression mechanisms, initially developed for the local setup, were successfully reused in FABRIC with minimal changes, demonstrating deployment-agnostic engineering practices (see Section 5.3 and Chapter 7).

[2] https://github.com/CollinPoetoehena/EnergyEfficiency_DYNAMOS

Chapter 2

Background

In this chapter, we provide the necessary background information for this thesis. First, a brief explanation of DYNAMOS is given. Next, microservices and related information will be presented. Finally, essential background information for the energy efficiency report pipeline introduced in this thesis will be provided.

2.1 Dynamically Adaptive Microservice-based Operating System (DYNAMOS)

DYNAMOS[1] is a self-adaptive middleware system that abstracts complex data exchange patterns into standardized microservices. The system is designed to dynamically compose these microservices to meet various policy and user requirements. The 'operating system' within DYNAMOS allows the optimization of extra functional properties, allowing energy savings based on the environment of the system [5].

A central feature of DYNAMOS is its ability to dynamically select archetypes and architectures in response to policies, user inputs, and system events, offering significant flexibility and self-adaptivity [9]. These dynamic compositions, referred to as 'microservice chains', are structured as directed acyclic graphs (DAGs)—graph structures where nodes (microservices) are connected in a way that allows data to flow in one direction without forming cycles. This ensures that each data exchange process follows a clear, non-redundant path from input to output. The chains are executed as ephemeral 'jobs', processing data exchange requests through the designated parties and ensuring compliance with the relevant policies [5, 9].

DYNAMOS is inspired by the architecture of Digital Data-sharing Marketplaces (DDMs), which are platforms where multiple parties securely exchange data under strict policy control, ensuring sovereignty over data assets. A key concept within DDMs is the use of archetypes—common data-sharing patterns that define how data is exchanged, including who provides the data, who processes it, and where results are delivered, all according to agreed-upon policies [5, 10]. For instance, one archetype may involve data being processed by a trusted third party before reaching the end user, while another may involve computation occurring directly at the data source. There are 11 archetypes commonly referenced in the literature (see Figure 2.1).

[1] https://github.com/Jorrit05/DYNAMOS

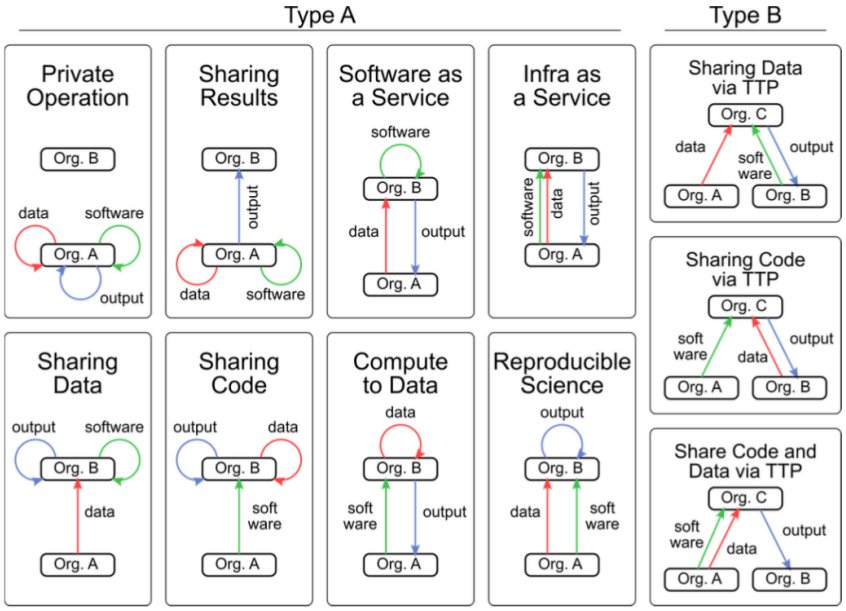


Figure 2.1: Data-sharing Archetypes [5, 10]

In DYNAMOS, a policy enforcer ensures adherence to these policies, determining which archetypes can be used and specifying parameters such as the location of data exchange and access permissions. DYNAMOS operationalizes archetypes by automatically composing and orchestrating microservices to fulfill data-sharing requests without manual intervention, enabling seamless integration of policies, archetypes, and both functional and extra-functional properties [5].

In practice, DYNAMOS uses extendable algorithms to create dynamic microservice chains tailored to each request. For example, the system may implement a 'Compute to Data' archetype—where computation is moved to the data source—or a 'Data through Trusted Third Party (TTP)' archetype to securely mediate data exchange. This dynamic capability enhances flexibility and ensures that DYNAMOS can adapt to diverse demands and environmental factors across distributed systems [5, 9].

2.1.1 UNL Use Case

This thesis uses the Universities of the Netherlands (UNL) use case from the AMdEX project[2] as the foundation for identifying root causes and validating energy optimizations. In this scenario, a data analyst conducts analyses on wage information from various universities across the Netherlands [5]. While the detailed context of the UNL use case is outside the scope of this thesis, its core relevance lies in the two archetypes it employs within DYNAMOS: the 'Data through Trusted Third Party (TTP)' and 'Compute to Data' archetypes. These archetypes serve as the main basis for the experimental validation of energy efficiency optimizations developed and tested throughout the thesis.

2.2 Microservices

The Microservice Architecture (MSA) is one of the most prevalent architectural styles for enterprise systems, designed to enhance scalability and maintainability by breaking down applications into smaller, loosely coupled components known as microservices. These microservices typically align with specific business domains, facilitating continuous delivery and deployment [3]. This architectural style offers several advantages, including improved maintainability, greater agility in development and delivery, and independent deployability and scalability [4].

By contrast, a monolithic architecture[3] involves building the entire application as a single, unified codebase. While monolithic systems are generally simpler to develop, test, and deploy in the early stages, they become increasingly difficult to manage as they grow. Larger monoliths tend to be tightly coupled, making them harder to maintain and scale effectively. MSA addresses many of these challenges

[2] https://amdex.eu/news/sharing-research-data-under-ones-own-conditions/
[3] https://www.atlassian.com/microservices/microservices-architecture/microservices-vs-monolith

by decoupling system components, allowing teams to develop, deploy, and scale services independently. However, it also introduces complexities in testing, debugging, and deployment, and typically requires developers to have a deeper understanding of distributed systems [11].

2.3 Green Computing

Green computing refers to the environmentally conscious design, use, and disposal of computing resources, with the goal of minimizing energy consumption, reducing e-waste, and lowering the overall environmental footprint of IT systems. As Information Communication Technology (ICT) has expanded exponentially, energy use and electronic waste have grown in parallel, making sustainability a pressing concern in both research and practice [12].

Core practices of green computing include designing energy-efficient hardware and software, recycling and repurposing outdated components, and minimizing resource use during manufacturing and operation. Recent developments also highlight advances in cloud computing efficiency, better hardware utilization, and sustainable management of Internet of Things (IoT) devices, which continue to grow rapidly in both scope and scale [12].

A key dimension of green computing is the role of software in driving sustainability. According to Balanza-Martinez *et al.* [6], green software can be divided into two categories. The first, *green in software*, focuses on optimizing software to reduce its own energy consumption. The second, *green by software*, refers to software that facilitates energy-efficient practices in other domains or systems. This distinction is crucial, as it highlights that software can either directly shrink its environmental footprint or act as a facilitator for broader sustainability.

In the context of this thesis, green computing principles are operationalized through a focus on improving energy efficiency within DYNAMOS. By implementing and empirically validating energy-efficient strategies, this research contributes to the broader objectives of green computing. Specifically, the thesis emphasizes *green in software*, targeting internal optimizations within DYNAMOS to achieve tangible reductions in energy consumption.

2.3.1 Clarifying Green Computing versus Energy Efficiency

Although often used interchangeably, green computing and energy efficiency are related but distinct concepts. Green computing is an umbrella term encompassing a wide range of environmentally friendly IT practices, including eco-conscious hardware design, responsible disposal methods, and sustainable manufacturing processes. In contrast, energy efficiency focuses on reducing energy consumption during the operational phase of a system [6, 12–14]. This thesis situates itself within the intersection of these domains by focusing on software-level energy efficiency—a critical subset of green computing—aimed at making DYNAMOS more sustainable in its operational lifecycle.

2.4 Energy Efficiency

IT energy consumption has become an increasingly important issue. In the past, addressing energy consumption was primarily the responsibility of hardware designers. However, as hardware capabilities have expanded, the impact of software behavior on energy consumption has also grown considerably [13, 14]. In practice, while the hardware in any programmable device ultimately determines the energy usage, it is the software that dictates how that energy is consumed [14]. In other words, the hardware is the direct consumer of energy, while the software drives its energy consumption [6].

According to Balanza-Martinez *et al.* [6], energy efficiency is about minimizing the energy consumption, and energy efficiency optimizations are focused on reducing energy consumption. Ardito *et al.* [14] state that developers should apply optimization strategies iteratively, making adjustments as needed. These approaches should be implemented with care, taking into account the software's mission, main functionalities, required quality of service, and the interests of stakeholders. For example, using an energy efficiency optimization could enhance energy efficiency, but it might also conflict with requirements like response time or availability.

The following sections outline the key areas where energy efficiency optimizations can be targeted. These definitions establish a foundation for identifying and implementing energy efficiency improvements explored in this thesis.

2.4.1 System and Software Levels

The system and software stack consists of various levels, each offering distinct opportunities to enhance the energy efficiency of software systems [6]:

- **Hardware**: At this level, optimizations are directed toward improving the energy efficiency of physical components by optimizing resource utilization [6].
- **Low-Level Software**: This level focuses on enhancing the efficiency of machine code through compiler optimizations, which transform source code into optimized executable code [6].
- **Operating System**: Optimizations at this level target energy consumption management by fine-tuning operating system functionalities, such as putting idle resources into sleep mode or efficiently scheduling resource usage [6].
- **Application Software**: This level emphasizes optimizing software designed for end-users. It focuses on energy-efficient application development by leveraging information unique to the application context, which lower system levels cannot access [6].

2.4.2 Platforms

Software applications can target different platforms. Balanza-Martinez *et al.* [6] identified the following platforms for energy efficiency optimization tactics:

- **Agnostic**: Tactics that are independent of the platform, such as selecting the most energy-efficient thread-safe data structure for Java applications [6].
- **Workstation**: Strategies tailored to single workstations, including desktops or servers, such as measuring the energy consumption of a single machine's CPU [6].
- **Distributed**: Techniques designed for distributed environments, such as employing cloud architectural patterns to minimize energy usage in cloud-native applications [6].
- **Mobile**: Optimizations specific to mobile devices, for example, grouping sensor requests within an application to reduce device energy consumption [6].

2.4.3 Abstraction Levels

Energy optimization techniques can target various abstraction levels within software applications [6]:

- **Architectural Level**: Representing the highest level of abstraction, this level addresses overarching software components, layers, their interactions, and the broader context. Decisions made at the architectural level define the fundamental structural organization of a software system. For example, using the most energy efficient programming language, software libraries and development environments [6].
- **Design Level**: This level focuses on decisions smaller in scope than architectural considerations but broader than language-specific idioms. While design-level choices do not alter the fundamental structure of a software system, they can significantly impact the architecture of subsystems. An example includes the adoption of specific design patterns [6].
- **Code Level**: The lowest level of abstraction, dealing with implementation specifics at the source code level. It focuses on how individual components and their relationships are realized using programming language features. For example, code refactoring techniques are applied at this level [6].

2.4.4 Distinction between Energy & Power

Energy is the capacity to perform work or induce change and is typically measured in joules (J) or kilowatt-hours (kWh). It can exist in various forms, such as kinetic, thermal, chemical, and electrical energy. Power, by contrast, refers to the rate at which energy is transferred or converted, expressed in watts (W), where one watt equals one joule per second. In simple terms, energy quantifies the total amount of work done, while power indicates how quickly that work is performed [15].

In the context of computing systems, this distinction is essential for understanding both performance and environmental impact. Systems such as Intel's RAPL (Running Average Power Limit) typically report power consumption in real-time, providing data on the instantaneous rate of energy use by components like CPUs, GPUs, and DRAM [16]. However, what ultimately matters for environmental sustainability and operational cost is the total energy consumed over a period of time. Because many

monitoring tools report only real-time power, energy consumption must often be estimated by integrating these power readings over time—a method used by systems like Kepler [17].

This thesis focuses on measuring and optimizing energy consumption rather than just power. While power metrics are useful for identifying immediate inefficiencies, it is the cumulative energy use that determines long-term environmental and operational impact, such as carbon emissions. Understanding and applying this distinction is crucial for evaluating the effectiveness of energy optimizations within DYNAMOS.

2.4.5 Measuring Energy Consumption

There are two common approaches to measure energy consumption [18]:

- **Hardware-based:** Measures power consumption directly and calculates energy consumption based on these values.
- **Software-based:** Estimates energy consumption using software tools and metrics.

A commonly used method involves retrieving multiple measurement points, each containing a timestamp and power value [18]. However, distributed cloud environments, like Kubernetes, present unique challenges. Virtual machines (VMs), often used in Kubernetes setups, typically do not expose real-time power consumption metrics [16, 19]. To overcome this limitation, Kepler was developed as a tool for estimating energy usage in such environments. Kepler's VM-based approach aligns with software-based estimation, while its Bare-metal (BM) implementation incorporates real-time hardware metrics, resembling a hardware-based approach. The energy consumption reporting mechanism of Kepler is discussed in detail in Section 3.1.3.

2.5 Energy Efficiency Report Pipeline

This section introduces the energy efficiency report pipeline developed for this thesis and explains key concepts that underpin its operation. The pipeline is implemented in Python[4], a widely used programming language known for its simplicity, versatility, and strong support for scientific computing. Python's combination of high-level data structures and a clear, object-oriented approach makes it particularly suitable for rapid development and data analysis [20].

Central to the pipeline are two main components: the Anomaly Detection (AD) algorithm and the Root Cause Analysis (RCA) algorithm. In computer science, an *algorithm* is defined as a finite, deterministic and effective problem-solving method suitable for implementation as a computer program [21]. These algorithms work together to identify inefficiencies in energy consumption within the system. The AD algorithm flags data points that deviate from normal behavior, while the RCA algorithm investigates and explains the underlying root causes for these deviations. In microservice-based systems, such tools are essential for maintaining performance and optimizing energy use [1].

In the pipeline developed for this thesis, energy consumption data is monitored, and the AD algorithm detects irregularities by comparing observed behavior to expected patterns. When anomalies are identified, the RCA algorithm is triggered to determine the underlying causes of abnormal energy usage. The selection and implementation of these algorithms are informed by the study of Floroiu *et al.* [1], which evaluates various AD and RCA techniques specifically for energy consumption in microservices. Their evaluation is based on three key metrics: correctness (precision), completeness (recall), and accuracy (F-score)—summarized in Table 2.1.

[4] https://www.python.org/

Terminology	**Metric Used**	**Definition**
Correctness	Precision	The proportion of true positive identifications among all positive identifications made by the algorithm [1]
Completeness	Recall	Shows how well the AD process can find relevant results, i.e., services that show energy anomalies [1]
Accuracy	F-score	Mean of precision and recall, measures the accuracy and completeness of the instruments [1]

Table 2.1: Terminologies and Metrics Used by Floroiu *et al.* [1]

An important aspect of anomaly detection is setting the threshold—a predefined value that determines when a data point is considered anomalous. This threshold directly affects the sensitivity of the detection process. Closely tied to this is the concept of calibration, which involves fine-tuning algorithm parameters to improve performance on specific datasets. Proper calibration ensures that the algorithm balances sensitivity and specificity, resulting in accurate and reliable detection outcomes [1]. Further technical details, including the specific implementation of the AD and RCA algorithms, are provided later in this thesis.

Chapter 3

Energy Consumption Root Causes

In this chapter, we identify the root causes of energy consumption in DYNAMOS, forming the foundation for the energy efficiency optimizations that will be discussed in Chapter 4. By analyzing energy measurements from targeted experiments and using tools such as Prometheus and Kepler, we pinpoint the main contributors to energy usage. A systematic investigation is applied to containers and processes within realistic data exchange operations.

The chapter begins with an overview of the energy measurement setup and preliminary findings used to identify containers of interest. It then presents a series of focused experiments to uncover the primary sources of energy consumption, supported by prior research and system-level traces. The results are categorized into distinct cause areas, providing a clear rationale for the optimization strategies introduced in subsequent chapters.

This structured analysis establishes a solid basis for improving the energy efficiency of DYNAMOS.

3.1 Energy Efficiency Report Pipeline

This section presents the conceptual implementation of the energy efficiency report pipeline, specifically adapted for DYNAMOS. The pipeline builds on the research conducted by Floroiu *et al.* [1] and is designed to facilitate the identification of root causes of energy consumption in DYNAMOS.

The first part details the implementation of the pipeline's phases, following the design described in Floroiu *et al.*'s [1] work on anomaly detection and root cause analysis for energy consumption in microservices. The process begins with energy consumption measurements, which serve as inputs for an anomaly detection (AD) algorithm. Subsequently, a root cause analysis (RCA) algorithm pinpoints the root causes of energy consumption.

The second part focuses on how the pipeline is implemented within DYNAMOS, highlighting the specific adaptations made to align with its design.

3.1.1 Energy Consumption Measurements

The first step in the energy efficiency report pipeline involves collecting energy consumption metrics—quantitative measurements that reflect how much energy is used by system components over time [22]. These metrics form the foundation for anomaly detection and root cause analysis in later stages of the pipeline.

To monitor energy usage effectively in a microservice-based system like DYNAMOS, Floroiu *et al.* [1] leverage the widely adopted open-source tools Prometheus[1] and Scaphandre[2]. Prometheus is used to collect container-level metrics such as CPU and memory utilization, while Scaphandre captures power consumption data and exports it in a format compatible with Prometheus. This integration allows energy-related insights to be unified within a common metrics infrastructure. Section 3.1.2 provides additional technical details on the final chosen energy monitoring tool.

The monitoring stack is tightly integrated with Kubernetes, the platform used to run DYNAMOS [5]. Kubernetes is a portable, extensible, open-source system for automating the deployment, scaling, and management of containerized applications [23]. Prometheus is already embedded within DYNAMOS for metrics collection, aligning with its status as the de facto standard for observability in Kubernetes

[1] https://prometheus.io/

[2] https://hubblo-org.github.io/scaphandre-documentation/index.html

environments[3]. This native compatibility simplifies integration and enables the reuse of some existing monitoring infrastructure.

To ensure comprehensive coverage across the cluster, cAdvisor[4] is deployed as a DaemonSet[5] within the Kubernetes environment. This ensures that metrics such as CPU and memory usage are gathered for every container on every node, enabling fine-grained monitoring and analysis of resource utilization patterns across DYNAMOS services.

3.1.2 Energy Monitoring Tool

While Floroiu *et al.* [1] utilize Scaphandre for energy monitoring, their study does not consider alternative tools. Furthermore, their analysis is conducted in a Docker environment, whereas DYNAMOS is deployed in Kubernetes. Currently, the two most widely used energy monitoring tools are Scaphandre and Kepler [24]. Therefore, we consider these two tools for energy monitoring in our study.

3.1.2.1 Scaphandre and Kepler Overview

Scaphandre is a monitoring agent specifically designed for tracking energy consumption metrics[6]. It facilitates the measurement and understanding of energy consumption patterns in technological services [25]. Scaphandre integrates seamlessly with Prometheus[7], enabling real-time monitoring.

Kepler (Kubernetes-based Efficient Power Level Exporter) is another tool for energy monitoring, leveraging eBPF (extended Berkeley Packet Filter) technology [26] to track per-container energy consumption efficiently. Kepler probes CPU performance counters and kernel tracepoints, using these metrics along with machine learning models to estimate energy consumption accurately. By integrating with Kubernetes, Kepler supports energy-efficient practices aligned with the Greenhouse Gas (GHG) Protocol and ensures fair power distribution [16, 17, 27–30].

3.1.2.2 Monitoring Tool Selection: Kepler versus Scaphandre

For this study, Kepler was selected as the primary energy monitoring tool due to its superior accuracy in container-level energy metrics, which aligns closely with the objectives of our pipeline. The pipeline aims to detect inefficiencies by pinpointing specific containers or services within DYNAMOS that are responsible for high energy consumption. As shown in the comparative study by Akbari *et al.* [31], Kepler outperforms Scaphandre in measuring container-level energy usage, making it particularly well-suited to Kubernetes-based microservice environments. In contrast, Scaphandre, while effective at collecting host-level metrics, tends to underestimate memory-related energy consumption at the container level—a limitation for detailed root cause analysis.

While Kepler has advantages in container-level monitoring, multiple studies have shown that both tools produce highly comparable results in practical settings. Centofanti *et al.* [32] evaluated Kepler and Scaphandre in Kubernetes clusters and reported similar outcomes even under varying workloads, concluding that either tool is suitable for cloud-based monitoring, particularly on bare-metal servers. Similarly, Gudepu *et al.* [33] found that both tools offer comparable monitoring accuracy despite differences in design. These findings suggest that while Kepler is more tailored to our use case, the overall choice of tool has a minimal effect on measurement accuracy.

In addition to its container-level precision, Kepler offers practical advantages in virtualized environments. Unlike Scaphandre, which depends on low-level hardware interfaces like `powercap`, Kepler supports energy estimation via pre-trained power models when direct access to hardware metrics is unavailable. This makes Kepler especially effective on cloud platforms that use virtual machines (VMs) such as AWS and Azure, where physical power sensors are often inaccessible [16, 19, 28]. In bare-metal Kubernetes clusters, Kepler can still use real-time metrics for greater accuracy.

This flexibility proved critical during local development and testing, where DYNAMOS was deployed on a Windows machine using Windows Subsystem for Linux 2 (WSL2)[8] in combination with Docker Desktop[9] to manage Kubernetes. In this setup, Scaphandre could not run due to missing kernel modules

[3] https://kubernetes.io/docs/concepts/overview/components/
[4] https://github.com/google/cadvisor
[5] https://kubernetes.io/docs/concepts/workloads/controllers/daemonset/
[6] Scaphandre metrics
[7] https://hubblo-org.github.io/scaphandre-documentation/tutorials/kubernetes.html
[8] https://learn.microsoft.com/en-us/windows/wsl/
[9] https://docs.docker.com/desktop/features/kubernetes/

like `powercap`, which are not exposed in WSL2 environments. This incompatibility, also reported in community discussions[10], is shown in Figure 3.1.

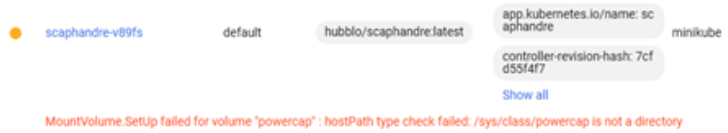


Figure 3.1: Scaphandre Incompatibility with Certain VM Environments

Considering the stronger support for container-level metrics, adaptability to virtualized environments, and practical compatibility with WSL2, Kepler emerged as the most effective and versatile monitoring tool for this research. A detailed explanation of Kepler's energy estimation methodology is provided in Section 3.1.3.

3.1.3 Kepler Energy Consumption Reporting

Kepler is a container-level energy monitoring tool designed for Kubernetes-based environments. As illustrated in Figure 3.2, its architecture consists of two main components: the Kepler Exporter, which handles resource monitoring and energy estimation, and an optional Model Server, which supports the training of custom power models [16, 19, 28].

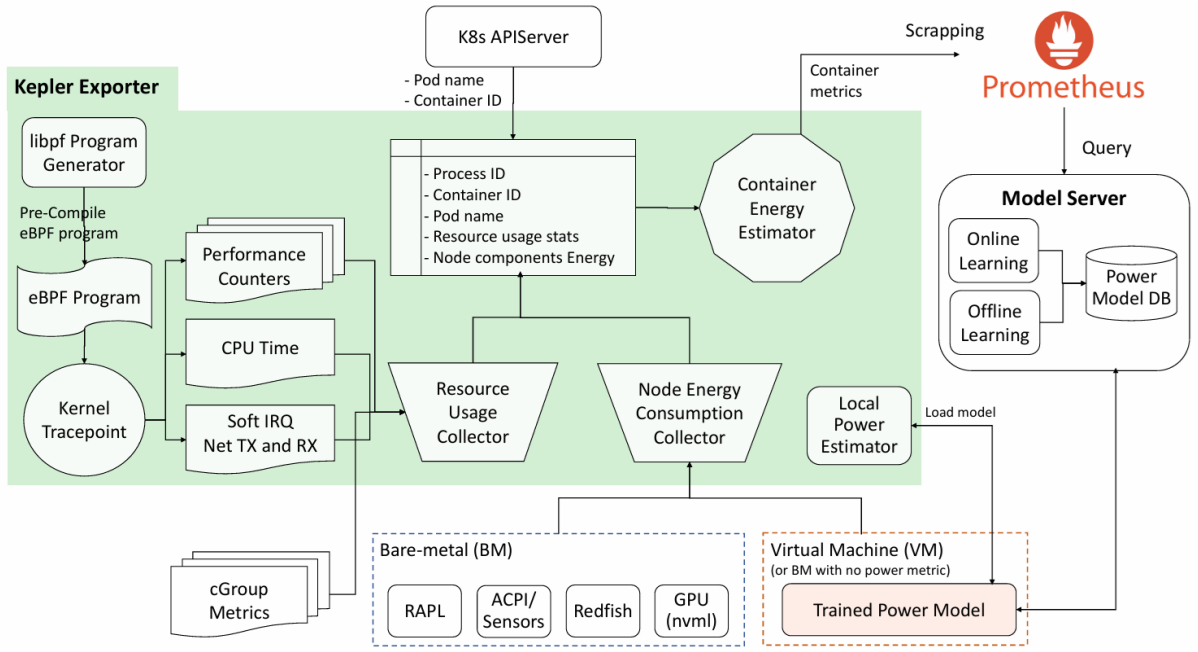


Figure 3.2: Kepler Architecture [16]

The Kepler Exporter uses eBPF[11] instrumentation integrated into the Linux kernel to collect process-level resource usage metrics—such as CPU time and hardware counters—with high granularity and minimal overhead. These metrics are combined with power measurements to estimate container-level energy consumption, which is then exported to Prometheus for observability and analysis [16, 17, 19, 28].

Kepler operates in two modes depending on the deployment environment: bare-metal (BM) or virtual machines (VMs). In BM environments, Kepler accesses real-time hardware metrics through standard interfaces such as RAPL (for CPU and DRAM) and NVML (for GPU power consumption). In VM environments, where such hardware telemetry is typically unavailable, Kepler uses pre-trained power models based on historical benchmark data to estimate energy consumption based on eBPF-collected

[10] https://github.com/hubblo-org/scaphandre/issues/383
[11] https://ebpf.io/

metrics like CPU usage and software interrupts [16, 17, 19, 28]. This distinction is illustrated in Figure 3.3.

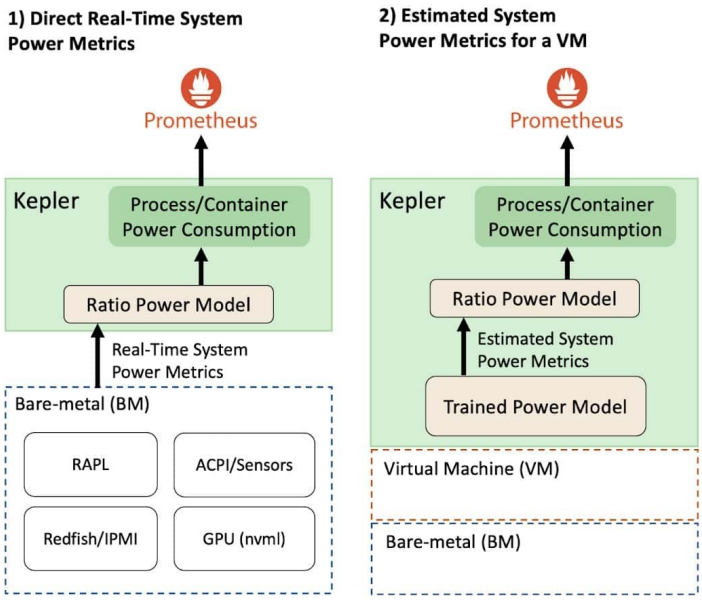


Figure 3.3: VMs versus BMs for Collecting Power Consumption Metrics [19, 28]

Kepler's energy estimation follows a two-step process. First, total system power—whether directly measured (in BM setups) or predicted via a power model (in VMs)—is divided into two components: dynamic power, which scales with system activity, and idle power, which remains constant regardless of workload. In the second step, dynamic power is attributed to containers based on their relative resource usage (e.g., CPU time), while idle power is allocated proportionally to container size, in accordance with the Greenhouse Gas (GHG) Protocol [17, 28]. This dual attribution mechanism allows Kepler to fairly and transparently estimate energy usage at the container level. The resulting per-container metrics are periodically exported to Prometheus, enabling fine-grained energy observability across Kubernetes workloads.

While Kepler's model-based VM mode lacks the precision of real-time telemetry, it offers a practical solution in virtualized and cloud environments. The pre-trained models, validated in several studies [16, 17], are built from benchmark data gathered in BM environments. Unlike traditional models that rely on aggregated system-level metrics, Kepler employs a process-level training strategy, using fine-grained performance data to estimate incremental energy consumption. This approach enables accurate container-level energy attribution, even in scenarios where workloads cannot be isolated for direct measurement [16, 17, 19, 28].

Nonetheless, pre-trained models do have known limitations. They may overestimate VM-level consumption, struggle to allocate idle power fairly in multi-tenant environments, and rely on accurate reporting from the hypervisor. Despite these constraints, they remain sufficiently accurate for detecting relative energy inefficiencies across containers and guiding optimization strategies [16, 17].

In this thesis, only the Kepler Exporter is deployed. The Model Server is omitted due to technical constraints associated with the VM-like WSL2 development environment, which lacks access to real-time hardware power data. Furthermore, training and validating custom power models was not feasible within the scope of this thesis. Instead, DYNAMOS uses Kepler's default pre-trained model embedded within the Exporter to estimate container-level energy usage.

3.1.4 Anomaly Detection

After gathering energy consumption metrics, it is essential to detect deviations or abnormalities. For this purpose, an AD algorithm is employed. The study by Floroiu *et al.* [1] demonstrated that, on average, Balanced Iterative Reducing and Clustering Using Hierarchies (BIRCH) is the most effective AD algorithm among those considered for both the Sock Shop and UNI-Cloud use cases, across various time windows and metrics (see Table 2.1 and Figure 3.4). Consequently, this thesis utilizes the BIRCH

algorithm for anomaly detection.

	Algorithm	TW5			TW10			TW30			TW60		
Sock Shop	BIRCH	1	2	2	1	1	1	1	1	1	1	1	1
	iForest	3	3	4	2	2	2	2	3	2	1	1	1
	KNN	1	1	1	2	2	2	2	3	2	1	1	1
	LSTM	2	2	3	1	1	1	1	2	1	1	1	1
	SVM	1	2	2	2	2	2	1	2	1	1	1	1
	Metrics	*P*	*R*	*F*	*P*	*R*	*F*	*P*	*R*	*F*	*P*	*R*	*F*
UNI-Cloud	BIRCH	1	1	1	1	1	1	1	1	1	1	1	1
	iForest	2	2	2	1	1	1	2	2	2	2	2	2
	KNN	1	1	1	2	2	2	2	2	2	2	2	2
	LSTM	2	2	2	2	2	2	2	2	2	2	2	2
	SVM	1	1	1	1	1	1	1	1	1	1	1	1
	Metrics	*P*	*R*	*F*	*P*	*R*	*F*	*P*	*R*	*F*	*P*	*R*	*F*

Figure 3.4: AD Algorithms Ranking [1]

Effective anomaly detection requires careful calibration of the algorithms. Proper parameter tuning significantly impacts the effectiveness of these algorithms. For instance, the BIRCH algorithm's efficacy decreases with larger intervals. While smaller intervals offer fine-grained data for detecting anomalies, they also increase computational load; conversely, larger intervals simplify the task but reduce effectiveness. Therefore, careful selection of the window size is essential for optimal performance [1].

3.1.4.1 Balanced Iterative Reducing and Clustering Using Hierarchies (BIRCH)

BIRCH is an algorithm designed for clustering large datasets with limited memory resources, minimizing I/O costs by scanning datasets only once. It uses Cluster Features (CFs) and a hierarchical CF tree structure to achieve fast and scalable clustering [34]. In simpler terms, BIRCH organizes data points into clusters using a hierarchical structure, similar to organizing topics in a library.

The core of BIRCH is the CF tree, where each node represents a CF, a compact summary of a cluster defined by three components:

$$\mathrm{CF}_1 = (N_1, \overrightarrow{LS_1}, SS_1)$$

Here, the CF contains three key components:

- **N**: Number of data points in the cluster.
- **LS**: Linear sum of the data points (i.e., sum of all the individual data points). The linear sum helps to calculate the average (mean) of the data points in the cluster. By dividing the linear sum by the number of data points, you get the mean.
- **SS**: Squared sum of the data points (i.e., sum of the squares of all the individual data points). The squared sum helps to calculate the spread (variance) of the data points in the cluster. Variance is a measure of how spread out the data points are from the mean.

This CF summary is efficient and accurate for clustering decisions. BIRCH handles very large datasets by summarizing data into compact CFs, often requiring only a single dataset scan. It supports incremental and dynamic clustering, allowing new data to be added without reprocessing the entire dataset, making it scalable and efficient for large-scale data [35].

3.1.5 Root Cause Analysis

After anomalies in energy consumption are detected, the next step is to identify their underlying causes. For this purpose, a Root Cause Analysis (RCA) algorithm is applied. This thesis adopts the Root Cause Discovery (RCD) algorithm, based on the comparative evaluation by Floroiu *et al.* [1], which demonstrated RCD's strong and consistent performance across multiple use cases and evaluation metrics.

The study evaluated various RCA algorithms in microservice-based environments, specifically using the Sock Shop and UNI-Cloud use cases. Performance was assessed using metrics such as Precision at k (PR@k), which measures the proportion of correct root causes among the top-k predictions. For example, PR@2 evaluates whether the true root cause is found within the top two predicted results. As shown in Figure 3.5, RCD consistently ranked highest on average across different time windows and scenarios.

	Algorithm	TW5			TW10			TW30			TW60		
Sock Shop	e-diagnosis	3	3	3	3	3	3	3	3	3	3	3	3
	MicroRCA	2	1	1	1	1	1	2	1	1	1	1	1
	RCD	1	2	2	2	2	2	1	2	2	2	2	2
	Metrics	*1*	*2*	*3*	*1*	*2*	*3*	*1*	*2*	*3*	*1*	*2*	*3*
UNI-Cloud	e-diagnosis	3	3	3	3	3	3	3	3	3	3	3	3
	MicroRCA	2	2	2	2	2	2	2	2	2	2	2	2
	RCD	1	1	1	1	1	1	1	1	1	1	1	1
	Metrics	*1*	*2*	*3*	*1*	*2*	*3*	*1*	*2*	*3*	*1*	*2*	*3*

Figure 3.5: RCA Algorithms Ranking [1]

Although RCD and MicroRCA appear comparable in terms of PR@k scores, Figure 3.6 shows that RCD demonstrates more consistent performance across different configurations. This figure presents the Mean Average Precision (MAP) for both algorithms using a 5-second time window. While MicroRCA performs slightly better in the Sock Shop case, RCD significantly outperforms it in the UNI-Cloud use case, highlighting its robustness across heterogeneous environments [1].

	Algorithm	Min	Max	Med	Mean	SD	CV
Sock Shop	**PR@1**						
	ϵ-diagnosis	0.0	1.0	0.0	0.04	0.064	0.054
	MicroRCA	0.0	1.0	0.0	0.582	0.493	0.244
	RCD	0.0	1.0	1.0	0.708	0.455	0.207
	PR@2						
	ϵ-diagnosis	0.0	1.0	0.0	0.08	0.091	0.08
	MicroRCA	0.0	1.0	1.0	0.918	0.274	0.076
	RCD	0.0	1.0	1.0	0.738	0.44	0.194
	PR@3						
	ϵ-diagnosis	0.0	1.0	0.0	0.15	0.357	0.128
	MicroRCA	0.0	1.0	1.0	0.996	0.066	0.004
	RCD	0.0	1.0	1.0	0.8	0.4	0.161
	MAP						
	ϵ-diagnosis	0.054					
	MicroRCA	0.832					
	RCD	0.749					
UNI-Cloud	**PR@1**						
	ϵ-diagnosis	0.0	1.0	0.0	0.071	0.257	0.066
	MicroRCA	0.0	1.0	0.0	0.229	0.42	0.177
	RCD	0.0	1.0	1.0	0.783	0.412	0.17
	PR@2						
	ϵ-diagnosis	0.0	1.0	0.0	0.146	0.353	0.125
	MicroRCA	0.0	1.0	1.0	0.692	0.462	0.214
	RCD	0.0	1.0	1.0	0.85	0.357	0.128
	PR@3						
	ϵ-diagnosis	0.0	1.0	0.0	0.433	0.496	0.247
	MicroRCA	0.0	1.0	1.0	0.692	0.462	0.214
	RCD	0.0	1.0	1.0	0.912	0.283	0.08
	MAP						
	ϵ-diagnosis	0.217					
	MicroRCA	0.538					
	RCD	0.849					

Figure 3.6: Root Cause Analysis, time window = 5s [1]

Another key advantage of RCD is its minimal configuration overhead. The study notes that MicroRCA required fine-tuning of parameters to achieve optimal results, whereas RCD delivered competitive results with default settings [1]. This makes RCD particularly suitable for time-constrained projects

like this thesis, where ease of integration and reliable performance are essential.

In summary, the RCD algorithm is selected for this thesis due to its consistent accuracy, minimal configuration requirements, and strong performance across diverse microservice-based workloads. These characteristics make it a reliable choice for identifying the root causes of energy inefficiencies in DYNAMOS.

3.1.5.1 Root Cause Discovery (RCD)

According to Ikram *et al.* [36], Root Cause Discovery (RCD) is a scalable and efficient method for root cause analysis in microservice-based systems. It employs a localized and hierarchical approach to quickly and accurately determine root causes. The localized approach focuses on smaller, relevant parts of a system rather than analyzing the entire system at once, while the hierarchical approach involves analyzing data at different levels of granularity, starting from broader system-wide metrics and progressively narrowing down to finer details. Figure 3.7 illustrates the hierarchical and localized learning algorithm of RCD. It follows a divide-and-conquer strategy by first splitting the dataset into small subsets to find interventional targets from each subset using the Ψ-PC algorithm (dashed boxes). Interventional targets are specific variables that, when changed, can impact the system's behavior. In the merge phase, RCD combines the candidate root causes from all subsets and recursively performs the same steps with the new set of variables. The process stops when the set of candidate root causes cannot be further reduced.

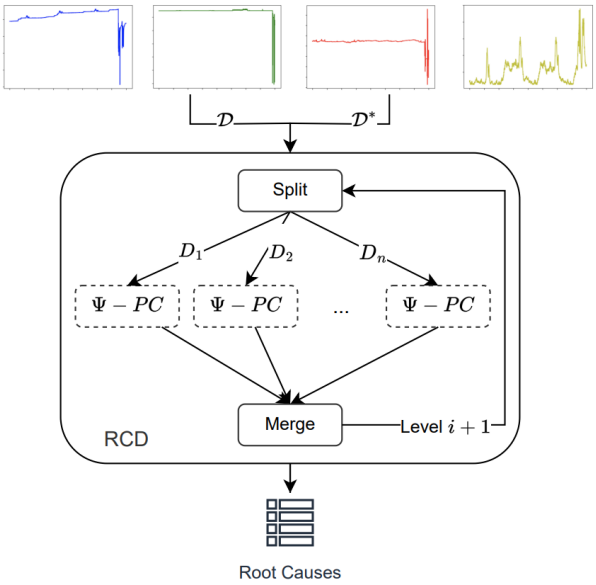


Figure 3.7: RCD Framework [36]

Figure 3.8 shows an execution of RCD with 11 nodes, where the orange nodes are potential root causes carried to the next level for further processing, and the red node (x_9) is the identified root cause. RCD estimates only the neighborhood of the F-NODE, a binary indicator (0 for normal periods and 1 for failure periods) that distinguishes between normal and failure periods, leaving the rest of the graph untouched, thereby significantly reducing the number of conditional independence (CI) tests. CI tests are statistical tests used to determine whether two variables are independent of each other given the presence of a third variable [36].

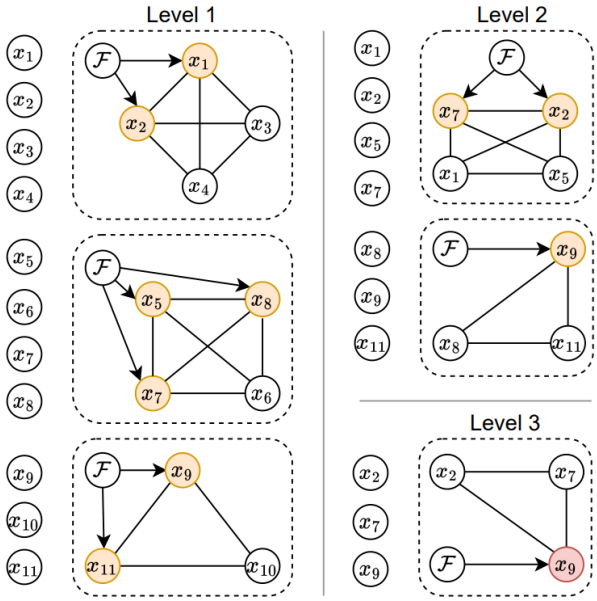


Figure 3.8: Execution of RCD with 11 Nodes and $\gamma = 4$ [36]

3.1.6 Implementation in DYNAMOS

This section explains the adaptation and implementation decisions and design of the 'energy efficiency report pipeline' in DYNAMOS. Figure 3.9 provides an overview of this pipeline.

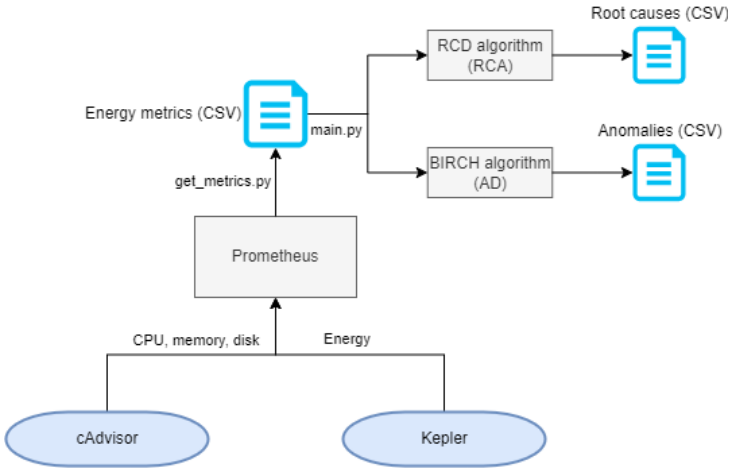


Figure 3.9: Energy Efficiency Report Pipeline Overview

3.1.6.1 Energy Metrics

In the study by Floroiu *et al.* [1], Scaphandre is used to gather energy consumption metrics. The following Prometheus query is employed to retrieve power consumption data:

```
"power" = f'sum(rate(scaph_process_power_consumption_microwatts[DURATION]))
                by (container_label_com_docker_compose_service) / 1000000"
```

Since power consumption is measured in Watts and energy consumption in Joules, Floroiu *et al.* [1] calculate energy consumption by converting the power consumption data. However, Kepler simplifies this process by directly exposing energy consumption metrics[12] [27]. The `kepler_container_joules_total` metric allows direct retrieval of energy metrics without conversion.

[12] https://sustainable-computing.io/design/metrics/#kepler-metrics-for-container-energy-consumption

The query used to fetch energy consumption metrics in the DYNAMOS code implementation, similar to the query with Scaphandre from Floroiu *et al.* [1, 27], is:

```
"energy" = f'sum(increase(kepler_container_joules_total[DURATION]))
                                    by (KEPLER_CONTAINER_NAME_LABEL)"
```

Here, `KEPLER_CONTAINER_NAME_LABEL` is a variable used in the code to distinguish containers in Kubernetes. The increase() function is used to calculate the total energy consumption in Joules over a specified time range, as opposed to rate(), which in this context would provide the power consumption.

3.1.6.2 CPU, Memory and Disk Metrics

The study by Floroiu *et al.* [1] utilizes cAdvisor[13] to gather CPU, memory, and disk metrics. In DYNAMOS' initial setup no metrics related to CPU, memory and disk were exposed. Consequently, cAdvisor had to be deployed as a separate DaemonSet in the Kubernetes environment to collect these metrics. Similarly to Floroiu *et al.* [1], CPU metrics are converted to percentages by multiplying the values by 100.

3.1.6.3 Time window

The study by Floroiu *et al.* [1] evaluates the algorithms using various time windows (5, 10, 30, and 60 seconds). The time window in this study corresponds to the scrape interval in Prometheus. Lower scrape intervals yield more data for analysis. However, in our initial local Docker Desktop setup cAdvisor required approximately 15 seconds on average to complete scraping all metrics in Prometheus due to the high volume of metrics it collects. Consequently, a 30-second time window (Prometheus scrape interval) was selected, ensuring that cAdvisor completes scraping before initiating a new cycle, while still maintaining an adequate amount of data for analysis.

3.1.6.4 Algorithms Usage

The RCD algorithm can be trained using a 'normal' dataset [1]. In other words, a dataset that contains the energy metrics with values that are considered normal behaviour. The RCD algorithm uses this dataset as a reference when detecting root causes. This dataset can be added in the data folder of the Python code before executing the algorithms in DYNAMOS.

However, the BIRCH algorithm does not require a separate training phase, as it directly uses the provided data to form clusters, which are then used to detect anomalies. The replicated study by Floroiu *et al.* [1] implemented the BIRCH algorithm using the following steps:

1. **Smoothing**: The data is smoothed using a rolling mean, which helps capture trends and reduces noise. In other words, data points are averaged to mitigate the impact of sudden outliers on different timestamps.
2. **Normalization**: The data is normalized, i.e., adjusted to a common scale, to ensure that all values are comparable.
3. **Clustering with BIRCH**: The CF tree is constructed in this step by invoking the fit function from the BIRCH model.
4. **Anomaly Detection**: Anomalies are identified based on the distance of data points from their nearest cluster center. Points that exceed a set threshold distance are classified as anomalies.
5. **Results Handling**: The results are saved, with each data point labeled as either an anomaly or normal, based on the BIRCH clustering output.

3.2 Container Selection for Energy Data Measurements

Before identifying the root causes of energy consumption, it is essential to select the containers to be included in the energy data measurements. Kubernetes environments typically contain numerous containers, many of which are not directly related to data exchange operations. Some containers may perform tasks unrelated to data exchange specifically, such as managing the Kubernetes environment itself. To ensure that energy measurements accurately reflect the energy usage of data exchange archetypes, only containers directly involved in these operations must be selected. However, this selection process proved

[13] https://github.com/google/cadvisor

challenging due to the large number of containers present in Kubernetes. To address this, we divided the container selection into several distinct steps, detailed in the following sections.

3.2.1 Pre-filtering

Before experimenting with different containers, we applied a pre-filtering process to exclude containers that are irrelevant to data exchange operations specifically or can be omitted for other reasons.

Firstly, containers responsible for metrics and monitoring were removed, including `kepler-exporter`, `cAdvisor`, `kube-state-metrics`, `ot-collector`, `prometheus`, and `grafana`. Secondly, general Kubernetes management containers, such as `config-reloader` and `kube-controller-manager`, were excluded. Similarly, containers related to tracing and logging, such as `jaeger` and `linkerd`, were removed. `Linkerd` service mesh containers, such as `identity` (for managing service identity) and `destination` (for service discovery and routing), were excluded as they are also unrelated to data exchange operations specifically. Finally, the `alertmanager` container, used for alerting, was excluded since it does not specifically relate to data exchange operations as well.

Specific DYNAMOS containers were determined to be critical and should be included in the energy data measurements:

- Agents: `uva`, `vu`, `surf`
- Additional SQL containers: `sql.*`
- Policy and orchestration: `policy.*` and `orchestrator`
- Sidecar pattern and message passing: `sidecar` and `rabbitmq`
- Request approval container: `api-gateway`

However, some containers still remain uncertain. The `coredns` container, which handles networking, cannot yet be definitively excluded. Similarly, the `linkerd-proxy` and `nginx-ingress` containers, which are slightly related to networking, will be included in the experimental phase to determine their relevance. Finally, the `system_processes` container might also be significant for energy consumption measurements and therefore cannot be excluded with certainty at this stage.

3.2.2 Preliminary Experimental Setup & Design

In this section, we describe the preliminary experiments conducted to evaluate the reliability of different containers for energy measurements and determine their suitability for further experimentation within this thesis. These experiments serve as a foundation for consistent and accurate energy consumption analysis in DYNAMOS' Kubernetes environment.

It is important to emphasize that the primary goal was not to produce statistically significant results, but rather to develop and validate a reliable methodology for energy measurement tailored to DYNAMOS. By conducting these early experiments, we aimed to enhance our understanding of energy monitoring in containerized systems and refine our experimental setup for subsequent analyses. The insights gained here directly inform the design and execution of the energy optimization experiments presented later in this study.

3.2.2.1 Hardware

All experiments for this setup are executed on a Lenovo laptop plugged in to a power outlet. This laptop has a Windows OS running WSL2. Kubernetes is deployed in Docker Desktop. Full details are listed in Table 3.1.

Parameter	Value
Software	
Operating System (Original)	Microsoft Windows 11 Pro, Version 10.0.26100 Build 26100
Operating System (WSL)	Ubuntu 22.04.3 LTS
Linux Kernel	5.15.167.4-microsoft-standard-WSL2
Docker Version	4.36.0
Container Runtime	Docker (docker://27.3.1)
Kubernetes Version	v1.30.5
Kubernetes Node Capacity	14 CPUs, 16GB memory, 110 pods
Hardware	
Computer	Lenovo (System Model: 21MN003GMH)
Processor	Intel(R) Core(TM) Ultra 7 155U, 1700 Mhz (12 Core(s))
RAM Capacity	32GB

Table 3.1: Specifications of System Under Test (SUT) Local

3.2.2.2 Windows Subsystem for Linux (WSL)

Local development in DYNAMOS was performed on a Windows machine using Windows Subsystem for Linux 2 (WSL 2)[14], a feature that allows users to run Linux distributions natively on Windows without the need for dual-boot setups or full virtual machines.

WSL provides a compatibility layer that enables Linux binaries to run within a Windows environment. It is especially useful for developers who require access to Linux tooling while continuing to work within the Windows ecosystem [37].

The second generation, WSL 2, introduces significant improvements by running a real Linux kernel inside a lightweight utility virtual machine (VM). Each Linux distribution installed under WSL 2 operates as an isolated container within this managed VM. These distributions share several system components—including the network namespace, CPU, memory, kernel, and device tree—but maintain separate namespaces for processes (PID), mounts, users, control groups (cgroups), and their own `init` processes. This hybrid architecture offers a balance between system-level isolation and lightweight integration with Windows [37].

While WSL 2 delivers a near-native Linux experience on Windows, it is important to note that it behaves similarly to a virtual machine. As such, it does not expose low-level hardware interfaces—such as `powercap`—required by certain monitoring tools like Scaphandre, which limits compatibility in some energy monitoring scenarios.

3.2.2.3 Sequence of Experiments

Energy consumption was measured under two scenarios: (1) idle (no actions performed) and (2) active (data exchange operations performed), each over a fixed 2-minute period. The idle scenario was measured first, followed by the active scenario, and this sequence was repeated to enable direct comparisons between the two conditions. This approach helps determine whether a container should be included in the final energy measurements.

The 2-minute duration was chosen to ensure consistent energy data reporting. Variability in Kubernetes background processes can lead to fluctuating energy consumption data. By maintaining a fixed measurement period, we minimize discrepancies caused by background activity. For instance, if data exchange tasks are performed for 1.5 minutes in one measurement and 2 minutes in another, additional background processes during the extra 30 seconds could influence the results, making comparisons unreliable. Using a fixed time frame ensures equal conditions across measurements.

3.2.2.4 Data Exchange Operation

During the active stage, data requests were made seven times at 7-second intervals within the 2-minute period. This interval was selected to prevent request timeouts caused by some existing issues in DY-

[14] https://learn.microsoft.com/en-us/windows/wsl/

NAMOS (see Section 8.3.2). Approval requests were obtained before each experiment run and reused for subsequent data requests to maintain consistency. This is because approval requests are similar for each data request, only differing in its jobId.

To minimize external influences on energy data, the default DYNAMOS settings were used, with optional services such as the aggregate service disabled. Listing 3.1 shows the SQL query used for these data exchange operations.

```
SELECT DISTINCT p.Unieknr, p.Geslacht, p.Gebdat, s.Aanst_22, s.Functcat, s.
    Salschal as Salary FROM Personen p JOIN Aanstellingen s ON p.Unieknr = s.
    Unieknr LIMIT 30000
```

Listing 3.1: SQL Query for Experiment

The corresponding JSON body for the data request is provided in Listing 3.2.

```
{
    "type": "sqlDataRequest",
    "query": "SELECT DISTINCT p.Unieknr, p.Geslacht, p.Gebdat, s.Aanst_22, s.Functcat, s.
    Salschal as Salary FROM Personen p JOIN Aanstellingen s ON p.Unieknr = s.Unieknr LIMIT
    30000",
    "algorithm": "",
    "options": {
        "graph": false,
        "aggregate": false
    },
    "user": {
        "id": "12324",
        "userName": "jorrit.stutterheim@cloudnation.nl"
    },
    "requestMetadata": {
        "jobId": "jorrit-stutterheim-5dfcf11a"
    }
}
```

Listing 3.2: SQL Data Request JSON Body

Note: the latest version of DYNAMOS integrates an API gateway that merges approval requests and data requests into a single endpoint. However, for the purpose of these experiments, and in consultation with one of the core DYNAMOS developers, the system was reverted to an earlier configuration where approval and data exchange requests are handled separately. This modification was made specifically for this thesis to enhance the system's stability during energy measurements (see Section 8.3.2). Separating the requests reduces the computational and communication overhead associated with each interaction, as the approval logic is processed independently. Once approved, the resulting parameters can be passed directly to the data request. This more modular approach led to a slight but meaningful improvement in the overall reliability and consistency of DYNAMOS, which is critical for accurate energy consumption analysis.

3.2.2.5 Cool-down and Minimizing Background Processes

A 30-second cool-down period was applied before each experiment set (i.e., both scenarios in the sequence) to reset metrics and synchronize with Prometheus scrapes, as recommended by Cruz [38]. This ensures accurate and consistent energy data collection.

Furthermore, background processes were reduced to the bare minimum, which further minimizes external influences [38]. Non-essential applications such as browsers and Postman were closed, and only the Prometheus UI, necessary for querying energy data, remained active in the Kubernetes environment. Other non-essential services, such as the Kubernetes Dashboard or Jaeger UI, were also disabled during the experiments to avoid interference.

3.2.3 Preliminary Experiment Results

To determine whether containers still under consideration should be included in the energy measurements, we conducted experiments as outlined in Section 3.2.2. The results are presented in Tables 3.2, 3.3, 3.4, and 3.5.

It is important to note that these results are preliminary and may differ from the final results presented in Chapter 7. These experiments were conducted as an initial evaluation to decide which containers to include for identifying energy consumption root causes and to gain a better understanding of how to measure energy consumption in Kubernetes environments. As Cruz [38] explains, several external factors (even room temperature) can affect energy measurements. The more detailed experimental setup described in Chapter 6 accounts for these factors to ensure greater reliability in the final results.

A minor form of anomaly detection was applied in this analysis, specifically to remove spikes in energy consumption values. Only three anomalies were identified and subsequently excluded from the results. No further data analysis or anomaly detection was performed, as the primary objective of these experiments was not to identify statistically significant changes but rather to gain insights and refine our approach, as outlined in Section 3.2.2.

3.2.3.1 `coredns` Container

The `coredns` container is a flexible DNS server used as the default DNS service in Kubernetes clusters. It resolves DNS queries for both internal and external resources. Table 3.2 showcases the results for the `coredns` container. Interestingly, energy consumption during active data exchange actions was consistently lower than during idle periods.

This behavior is likely due to the nature of the container's idle-state operations. When idle, the container performs background tasks, such as updating DNS records and conducting periodic health checks for service endpoints. However, during data exchange actions, the workload may shift away from these background tasks to handling actual DNS queries, which might be computationally lighter. Due to time constraints, this hypothesis was not further validated. Future experiments could involve disabling health checks and repeating measurements to determine whether these background tasks are indeed responsible for the observed energy savings during active periods.

We chose to exclude the `coredns` container from the final energy measurements, as its functionality is unrelated to data exchange operations. Including it would risk introducing variability (i.e. noise [39]) caused by unpredictable background processes, reducing the reliability of the results [39].

Run	Idle Energy(J)	Active Energy(J)	Difference(J)
1	127.737	102.333	-25.404
2	127.677	104.787	-22.89
3	120.738	105.456	-15.282
4	125.224	104.862	-20.362
5	130.554	108.322	-22.232
6	126.555	101.482	-25.073
7	128.307	106.412	-21.895
8	121.924	106.844	-15.080
9	124.893	105.328	-19.565
10	131.468	109.097	-22.371
Avg	126.508	105.492	-21.015

Table 3.2: Energy Consumption Experiment `coredns` Container

3.2.3.2 `linkerd-proxy` Container

The `linkerd-proxy` container forms the data plane of the Linkerd service mesh and is a lightweight, high-performance network proxy. It manages incoming and outgoing service traffic while providing features such as load balancing, traffic routing, and security via mutual TLS [40, 41]. Its focus is primarily on internal communication within the Kubernetes cluster, facilitating service-to-service communication rather than external traffic.

Table 3.3 showcases the energy consumption results for the `linkerd-proxy` container. Similar to the `coredns` container, energy consumption was consistently lower during active data exchange operations compared to idle periods. This behavior can likely be attributed to background tasks such as connection management, health checks, and keeping connections alive during idle states. When handling data

exchange traffic, the workload may shift to active tasks that are computationally more efficient, leading to reduced idle overhead. Due to time constraints, this hypothesis was not further validated. Future experiments could disable health checks and other background tasks to assess their impact on energy consumption, further supporting or refuting this explanation.

We chose to exclude the `linkerd-proxy` container from the final energy measurements because its primary role is unrelated to data exchange operations. The Linkerd service mesh is primarily used in DYNAMOS for observability, security, and reliability features, with distributed tracing being a key advantage through the Jaeger add-on [5, 42]. Including the `linkerd-proxy` container would increase the risk of unpredictable background processes influencing the measurements, reducing their reliability.

Run	Idle Energy(J)	Active Energy(J)	Difference(J)
1	169.998	134.616	-35.382
2	162.843	121.983	-40.86
3	150.276	125.946	-24.33
4	162.565	120.956	-41.609
5	160.432	123.784	-36.648
6	165.346	129.221	-36.125
7	157.889	127.046	-30.843
8	170.232	135.114	-35.118
9	163.568	123.765	-39.803
10	158.323	124.226	-34.097
Avg	162.147	126.666	-35.482

Table 3.3: Energy Consumption Experiment `linkerd-proxy` Container

3.2.3.3 `nginx-ingress` Container

Ingress provides a means to expose HTTP and HTTPS routes from external clients to services running within a Kubernetes cluster. The Ingress functionality is managed by an Ingress controller, such as `ingress-nginx`, which handles the routing and fulfillment of Ingress rules [43]. The `nginx-ingress` container serves as an Ingress controller in Kubernetes, managing and routing HTTP/HTTPS traffic between external clients and services within the cluster [43]. For example, it facilitates requests such as `/api/requestApproval` in DYNAMOS, which originates externally.

Table 3.4 shows the results for the `nginx-ingress` container. Similar to the `coredns` and `linkerd-proxy` containers, energy consumption was consistently lower during active data exchange actions. The likely explanation is equal to the explanation of the examined reduction for the `linkerd-proxy` container.

We chose to exclude the `nginx-ingress` container from the final energy measurements. Its role primarily involves external traffic and is less relevant to internal data exchange operations. Including it could introduce similar risks of measurement variability as observed with the `coredns` and `linkerd-proxy` containers.

Run	Idle Energy(J)	Active Energy(J)	Difference(J)
1	679.469	568.971	-110.498
2	811.557	712.964	-98.593
3	802.419	694.762	-107.657
4	870.376	755.196	-115.18
5	790.321	670.854	-119.467
6	845.632	734.170	-111.462
7	710.055	610.048	-100.007
8	883.421	771.036	-112.385
9	762.399	646.513	-115.886
10	795.778	686.659	-109.719
Avg	795.143	685.117	-110.085

Table 3.4: Energy Consumption Experiment `nginx-ingress` Container

3.2.3.4 `system_processes` Container

The `system_processes` container captured the largest energy consumption difference between idle and active states (see Table 3.5). This suggests that significant energy consumed during data exchange operations is reported by this container. However, it also introduces the risk of capturing energy consumed by unrelated background Kubernetes processes.

Excluding the `system_processes` container could result in incomplete measurements, as it reports energy from data exchange operations that might not be captured elsewhere. To mitigate variability, the experimental design described in Chapter 6 incorporates several counters, such as repeated measurements across multiple runs to ensure reliable data.

Based on these considerations, we included the `system_processes` container in the final energy measurements.

Run	Idle Energy(J)	Active Energy(J)	Difference(J)
1	21676.520	22273.421	596.901
2	21574.011	22227.11	653.099
3	21618.545	22298.915	680.37
4	21614.108	22322.240	708.132
5	21622.097	22313.987	691.89
6	21658.178	22347.073	688.895
7	21606.456	22274.861	668.405
8	21650.206	22324.497	674.291
9	21653.198	22313.987	660.789
10	21504.915	22090.480	585.559
Avg	21617.823	22278.657	660.833

Table 3.5: Energy Consumption Experiment `system_processes` Container

3.2.3.5 Conclusions

These experiments demonstrated the feasibility of using Kepler to produce reliable energy consumption data in Kubernetes environments. While energy consumption varied across containers, these variations were consistent for the same scenarios and multiple runs.

The experiments also validated the use of a 2-minute cool-down period, as we noticed that shorter intervals (e.g. 1 minute or 1.5 minutes) introduced slight inaccuracies, while longer periods showed no significant benefit while executing the experiments.

3.2.4 Challenges and Insights in Measuring Energy Consumption in Kubernetes

While experimenting with energy measurements, we encountered significant challenges in identifying the optimal set of containers to include. One primary difficulty was the variability in results across different measurements. Containers such as `coredns` and `linkerd-proxy` exhibited substantial variations in energy consumption, as shown in Tables 3.2 and 3.3. This variability is largely due to the influence of background processes in the Kubernetes environment, such as tracing, and other routine operations.

Another challenge was the inconsistency in how containers reported energy consumption. For example, energy usage was sometimes attributed to the `sql-query` container, while in other instances, it was recorded in the `uva` container. Similarly, energy consumption could be distributed across containers such as `uva`, `surf`, and `vu`, or only assigned to `system_processes`. This variability seems to arise from Kepler's method of attributing energy usage among containers. Kepler adheres to the GHG protocol, which divides power consumption across containers based on their allocated resources, such as container size, and resource utilization [16]. This approach likely explains why significant energy consumption is reported under the `system_processes` container, as this container aggregates metrics from the operating system, serving as the primary container in the Kubernetes environment.

Despite these minor inconsistencies in container-level reporting, we observed that the total energy consumption remained relatively consistent across measurements, irrespective of how it was distributed among specific containers. By ensuring that all relevant containers were included in the energy measurements, this variability had minimal to no impact on the overall accuracy of the total energy data found in our results in Section 3.2.3.

3.2.4.1 Prometheus Query for Energy Measurements

We experimented with two approaches for querying energy consumption data in Prometheus:

1. Using the `increase()` function[15], which estimates energy usage based on extrapolated (i.e. estimated) values.
2. Directly subtracting the total consumed energy at the start of the experiment from the total consumed energy at the end. In other words, measuring the total energy consumed at the start and the end of performing the data exchange operations.

Both methods rely on the most recent data scrapes in Prometheus; however, we found the first approach to be more reliable for energy measurements. The `increase()` function calculates energy consumption over an exact time range, ensuring consistent comparisons across measurements. By contrast, the second approach depends on the total energy consumed at specific moments, which may introduce slight variations due to differences in scrape timing. For example, if a new data scrape occurs during a measurement, the total energy value could increase slightly, causing discrepancies. This issue is especially pronounced with short scrape intervals, such as 5 seconds, where the risk of inconsistencies is higher. Conversely, longer scrape intervals reduce this risk, as new scrapes occur less frequently.

While we use a relatively longer scrape interval, as detailed in Section 3.1.6.3, the first approach remains preferable. Its reliance on exact time ranges minimizes the potential for inaccuracies, ensuring more consistent and reliable energy measurements. Consequently, we adopt the `increase()` function for all subsequent energy measurement queries.

3.2.4.2 Minimizing Background Processes

To establish a robust energy measurement setup for the data exchange operations, we minimized the number of containers included in our analysis. Containers with significant background processes, such as `nginx-ingress`, were excluded, leaving only those directly involved in data exchange operations, such as `system_processes`. This focused approach ensures more accurate energy measurements by reducing potential variability caused by unrelated activities.

Despite these efforts, background processes inherent to the Kubernetes environment remain a limitation. While we can mitigate their impact by carefully selecting containers, their influence on energy measurements cannot be entirely eliminated. This reflects a broader challenge of measuring specific aspects of energy consumption in Kubernetes environments, rather than focusing on total energy consumption across the entire system.

[15] https://prometheus.io/docs/prometheus/latest/querying/functions/#increase

3.2.5 Final Selected Containers Query

After thorough analysis, we finalized the set of containers to include in the energy measurements. The resulting Prometheus query is shown in Listing 3.3. This query filters for the selected containers via the `container_name` property in the Kepler energy metrics, and the data is grouped by `container_name`. Grouping enables analysis of both the individual containers' contributions and the combined energy consumed during experiments. This query could be used to gather energy measurements data for subsequent experiments.

```
sum(increase(kepler_container_joules_total{container_name=~"system_processes|uva|vu|surf|sql
    .*|policy.*|orchestrator|sidecar|rabbitmq|api-gateway"}[2m])) by (container_name)
```

Listing 3.3: Prometheus Energy Measurements Data Query

As discussed in Section 3.2.4, measuring the with the `increase()` function for an experiment provides the most accurate results. The time range specified in the query (e.g., `2m`) can be adjusted as needed, such as to 3 minutes (`3m`) or other intervals, depending on the experiment's duration.

Alternatively, the other approach described in Section 3.2.4 can also be used. The corresponding query is shown in Listing 3.4.

```
sum(kepler_container_joules_total{container_name=~"system_processes|uva|vu|surf|sql.*|policy
    .*|orchestrator|sidecar|rabbitmq|api-gateway"}) by (container_name)
```

Listing 3.4: Alternative Prometheus Energy Measurements Data Query

3.3 Identifying Root Causes

This section identifies the primary contributors to energy consumption in DYNAMOS. By leveraging the energy efficiency report pipeline and analyzing system traces and bottlenecks, key root causes are pinpointed. These findings form the foundation for selecting and implementing energy optimizations to address energy inefficiencies.

3.3.1 Energy Efficiency Report Pipeline

The energy efficiency report pipeline, described in Section 3.1, can be configured to determine the root causes of energy consumption. To narrow the analysis, specific containers are selected, as outlined in Section 3.2.

Using a setup similar to that described in Section 3.2.2, experiments were conducted to measure energy consumption during data exchange operations. The pipeline identified the `system_processes` and `sidecar` containers as the main contributors to energy usage. These results were manually verified using Prometheus energy data queries, as detailed in Section 3.2.5.

While the pipeline effectively highlights which containers consume the most energy, its interpretability varies depending on the container. In the case of the `sidecar` container, its function within DYNAMOS is well-defined and isolated, making it easier to understand the relationship between its behavior and energy consumption. By contrast, the `system_processes` container aggregates a large and diverse set of system-level processes. This aggregation makes it difficult to pinpoint which specific processes are responsible for high energy usage, limiting the actionable insights that can be derived from its data.

This limitation stems from Kubernetes' reporting model, where energy metrics are aggregated at the container and pod level. While this is useful in identifying high-level contributors, it lacks the granularity needed to attribute energy consumption to specific processes—especially within containers hosting numerous unrelated services.

To address this, the following section investigates time-intensive and potentially energy-intensive processes in more detail, aiming to identify finer-grained bottlenecks beyond container-level analysis.

3.3.2 Bottlenecks

Stutterheim [5] conducted a detailed investigation of bottlenecks in DYNAMOS. Figures 3.10 and 3.11 illustrate traces of SQL requests for the Compute to Data and Data through TTP archetypes for the UNL use case. The traces reveal that a significant amount of time is spent transferring data, particularly in the Data through TTP archetype. This is largely due to the number of transfers required by the

microservice architecture. As noted by Stutterheim, microservices inherently involve more data transfers compared to monolithic architectures, as services must exchange data to fulfill requests. Furthermore, considerable time is spent processing queries and retrieving data in both archetypes.

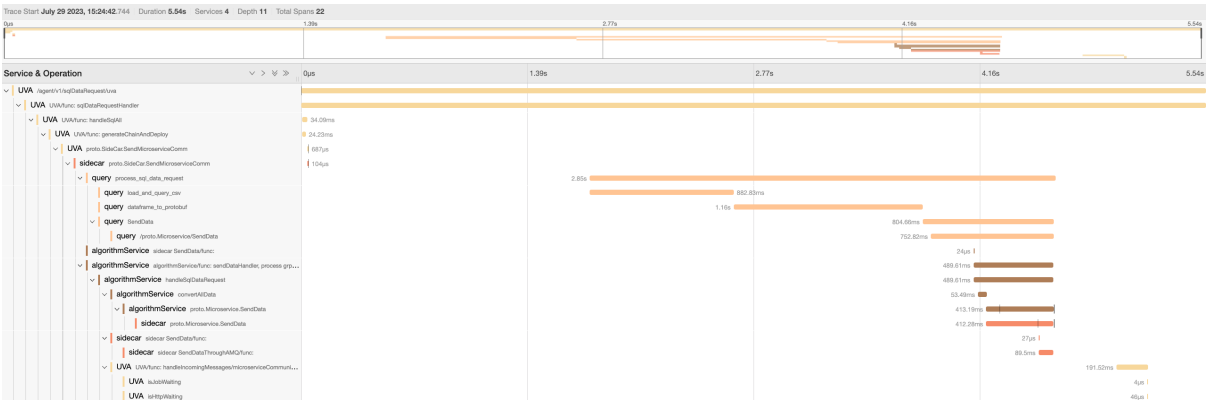


Figure 3.10: Trace of 30000 Line SQL Request, Compute to Data Archetype [5]

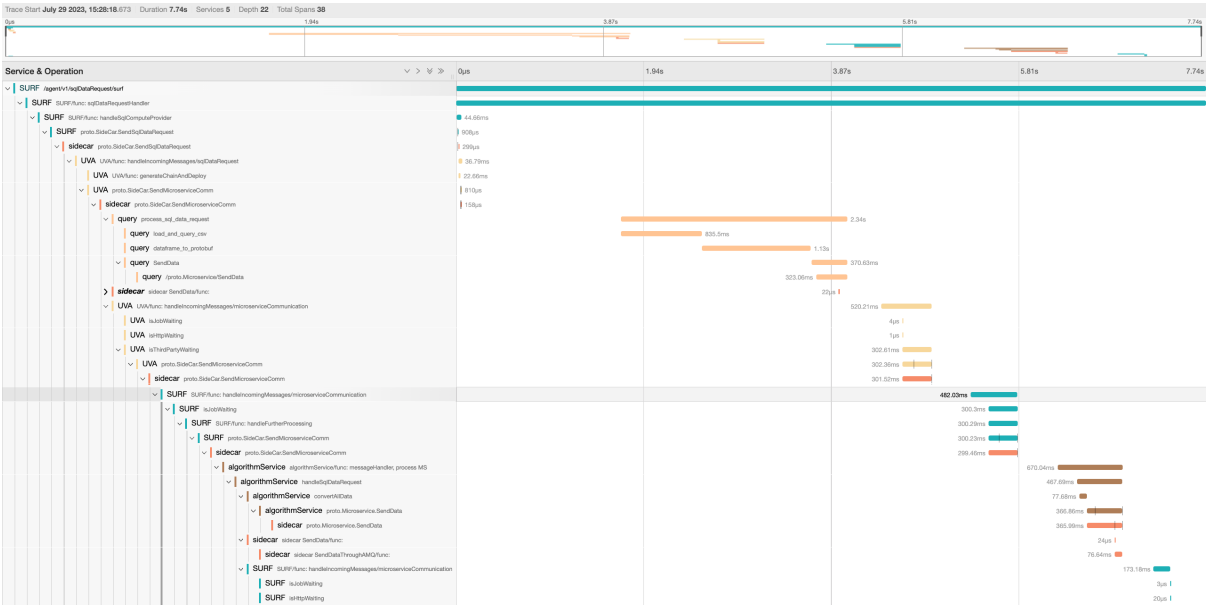


Figure 3.11: Trace of 30000 Line SQL Request, Data through TTP Archetype [5]

Figures 3.12 and 3.13 present the results of another experiment conducted by Stutterheim [5], which analyzed three key contributors to delay in the DYNAMOS system: startup time (initializing services), processing time (executing queries within the query and algorithm service), and transferring time (data exchanges between services).

As highlighted by Stutterheim, startup time, particularly during cold starts, is a significant bottleneck, with an average delay of 1.7 seconds. Processing time is also substantial, especially for larger queries such as those involving 30000 lines, aligning with observations from trace analysis. Lastly, transferring time is a major contributor, particularly in the Data through TTP archetype, where multiple service-to-service data transfers further amplify delays.

Limit	Latency	Startup Time	Total Processing Time	Transfer Times	Startup %	Processing Time %	Transfer Time %	Transfer Time % Excl. Startup
10000	3,22	1,55	1,15	0,52	48,23%	35,71%	16,06%	31,01%
10000	3,41	1,55	1,36	0,50	45,51%	39,88%	14,60%	26,80%
10000	3,17	1,57	1,14	0,47	49,50%	35,84%	14,67%	29,04%
10000	4,14	2,11	1,40	0,64	50,97%	33,70%	15,34%	31,28%
10000	3,73	1,97	1,22	0,54	52,73%	32,71%	14,56%	30,80%
20000	4,31	1,58	1,83	0,90	36,68%	42,41%	20,90%	33,02%
20000	5,63	2,16	2,43	1,05	38,37%	43,07%	18,56%	30,12%
20000	4,47	1,61	1,84	1,02	36,04%	41,19%	22,77%	35,61%
20000	4,42	1,60	1,95	0,88	36,11%	44,05%	19,84%	31,06%
20000	5,11	1,69	2,28	1,14	33,15%	44,56%	22,29%	33,34%
30000	5,24	1,54	2,37	1,33	29,41%	45,17%	25,42%	36,01%
30000	5,76	1,65	2,85	1,27	28,56%	49,45%	21,99%	30,78%
30000	5,34	1,70	2,42	1,22	31,84%	45,37%	22,79%	33,43%
30000	5,2	1,62	2,41	1,17	31,23%	46,31%	22,46%	32,66%
30000	5,41	1,57	2,62	1,22	28,96%	48,43%	22,61%	31,82%

Figure 3.12: Bottleneck Analysis, Compute to Data Archetype [5]

Limit	Latency	Startup Time	Total Processing Time	Transfer Times	Startup %	Processing Time %	Transfer Time %	Transfer Time % Excl. Startup
10000	4,13	1,73	1,076	1,33	41,84%	26,05%	32,11%	55,20%
10000	4,41	1,63	1,211	1,57	37,01%	27,46%	35,53%	56,41%
10000	4,49	1,61	1,299	1,58	35,88%	28,93%	35,19%	54,88%
10000	4,3	1,69	1,137	1,47	39,33%	26,44%	34,23%	56,42%
10000	3,99	1,62	1,084	1,28	40,65%	27,17%	32,18%	54,22%
20000	6,16	1,669	1,828	2,66	27,09%	29,68%	43,23%	59,30%
20000	6,41	1,785	1,771	2,85	27,85%	27,63%	44,52%	61,71%
20000	6,11	1,729	1,845	2,54	28,30%	30,20%	41,51%	57,89%
20000	5,65	1,589	1,606	2,46	28,12%	28,42%	43,45%	60,45%
20000	6,84	2,417	1,607	2,82	35,34%	23,49%	41,17%	63,67%
30000	7,66	1,616	2,271	3,77	21,10%	29,65%	49,26%	62,43%
30000	7,59	1,563	2,377	3,65	20,59%	31,32%	48,09%	60,56%
30000	7,7	1,525	2,183	3,99	19,81%	28,35%	51,84%	64,65%
30000	8,31	1,604	2,527	4,18	19,30%	30,41%	50,29%	62,32%
30000	7,99	1,758	2,218	4,01	22,00%	27,76%	50,24%	64,41%

Figure 3.13: Bottleneck Analysis, Data through TTP Archetype [5]

3.3.3 Conclusion of Root Cause Analysis

Based on the combined findings from the energy efficiency report pipeline and system-level trace and bottleneck analysis, three primary contributors to energy consumption in the DYNAMOS system were identified:

1. **Startup Time (RC1)**: The time required to initialize services, i.e. cold starts.

2. **Processing Time (RC2)**: The time spent executing SQL queries and retrieving data.
3. **Transferring Time (RC3)**: The time required for data transfers between services.

These root causes reflect the patterns observed in container-level energy measurements. For instance, the `system_processes` container likely captures energy used during service initialization and data processing, while the `sidecar` container is closely associated with data transfer operations—both of which were identified as dominant contributors in the energy efficiency report pipeline.

The findings are further supported by system trace and delay analyses, which consistently showed that startup overhead, processing time, and service-to-service communication significantly impact performance and, by extension, energy consumption.

Addressing these bottlenecks is expected to improve both system speed and energy efficiency. This aligns with conclusions drawn by Yuki and Rajopadhye [44], who highlight that faster systems are generally more energy-efficient due to reduced execution time. The root causes identified here serve as the basis for selecting and evaluating energy efficiency optimizations in the following chapters. Table 3.6 provides unique identifiers for each root cause, enabling easy reference throughout this thesis.

ID	Name
RC1	Startup Time (Cold Starts)
RC2	Processing Time (Query Execution)
RC3	Transferring Time (Data Transfers)

Table 3.6: Final Root Causes

Chapter 4

Energy Efficiency Optimizations

This chapter focuses on identifying and selecting energy optimizations applicable to data exchange archetypes in general, with a particular emphasis on their implementation in DYNAMOS. The chapter begins by outlining the selection process used to evaluate potential optimizations. Next, the candidate optimizations are discussed in detail. Finally, the chapter concludes with an explanation of the optimizations selected for implementation in DYNAMOS.

4.1 Optimizations Selection Process

The selection process for energy optimizations focuses on addressing energy inefficiencies in DYNAMOS, guided by both existing literature and an analysis of the system's root causes of energy consumption. While the literature offers numerous optimization strategies targeting different abstraction levels and platforms, as outlined in Section 2.4, it remains underdeveloped in the context of energy efficiency in software systems. This gap, highlighted by Balanza-Martinez *et al.* [6], necessitates the inclusion of potential optimizations derived from reasoning about DYNAMOS' specific root causes of energy consumption.

Given the time constraints of this thesis, a selective approach was adopted to limit the number of optimizations considered. The final set of selected optimizations was determined based on a combination of existing literature and the identified root causes of energy consumption in DYNAMOS (see Section 3.3.3).

To refine the initial list of candidate energy efficiency optimizations, a preliminary set of requirements, shown in Table 4.1, was established. We reduced the scope to nine candidate optimizations, detailed in Section 4.2. Requirement R1 ensures that the selected optimizations are applicable to data exchange archetypes, while Requirement R2 narrows the focus to optimizations targeting the root causes identified in Section 3.3.

ID	Requirement
R1	Optimizations must be relatable to data exchange archetypes
R2	Optimizations must focus on the identified root causes in Section 3.3

Table 4.1: Requirements Candidate Optimizations

After identifying candidate optimizations, an additional set of requirements, presented in Table 4.2, was applied to select optimizations that could be implemented in DYNAMOS. This set is limited to three potential optimizations that could be implemented in DYNAMOS. Requirement R3 restricts the selection to optimizations directly applicable to DYNAMOS' codebase and archetypes. Requirement R4 ensures that no optimization violates the design principles or requirements established during DYNAMOS' development, as outlined by Stutterheim *et al.* [5, 9]. Finally, Requirement R5 excludes optimizations that cannot be implemented within the available time, such as major architectural changes or reprogramming in a different language.

The final selection of optimizations was made based on comparisons of their suitability, potential impact on energy efficiency, and feasibility for implementation. These selected optimizations are described in detail in Section 4.3.

ID	Requirement
R3	Optimizations must be applicable to DYNAMOS' archetypes
R4	Optimizations must not violate the design of DYNAMOS (i.e. design requirements [5, 9])
R5	Optimizations must be implementable within the given time constraints

Table 4.2: Additional Requirements Selected Optimizations for Implementation

4.2 Candidate Energy Optimizations for DYNAMOS

This section presents the energy efficiency optimizations identified as candidates for data exchange archetypes.

4.2.1 Batch Processing

Batch processing involves processing data in groups rather than in real-time, optimizing resource usage by reducing overhead—i.e., the additional time, energy, or resources required to initiate, manage, and execute tasks unrelated to the core data processing itself. Furthermore, batching data requests and transmissions can lower energy consumption by minimizing network overhead, as fewer transmission events are needed [45–47]. In essence, batch processing eliminates redundant operations, resulting in more efficient resource utilization and improved performance.

4.2.1.1 Relation to Data Exchange Archetypes & Root Causes

Within the context of data exchange archetypes, batch processing consolidates multiple smaller operations into larger, more efficient tasks, reducing resource usage and energy consumption. This optimization directly addresses the following root cause:

- **Root Cause RC2:** By grouping data requests, batch processing allows queries to be executed collectively, potentially avoiding repeated processing of the same query.

The suitability of batch processing depends on the characteristics of the data exchange system. For instance, in systems that rely on immediate responses to individual requests, batch processing may not be practical. Conversely, in systems where requests can be aggregated and results are not time-sensitive, batch processing can be a highly suitable optimization.

4.2.2 Caching

Caching is a system design strategy that involves storing frequently accessed data in a location that is faster and easier to retrieve. The primary goal of caching is to enhance system performance and efficiency by reducing the time and resources required to access commonly used data [48]. Studies have demonstrated that caching can significantly improve energy efficiency [49, 50].

4.2.2.1 Relation to Data Exchange Archetypes & Root Causes

In the context of data exchange archetypes, caching improves energy efficiency by minimizing redundant operations and reducing system load and latency. Specifically, it addresses the following root causes:

- **Root Cause RC1:** Caching entire requests can eliminate the need to start up ephemeral services, such as those in DYNAMOS, by immediately returning cached responses to users without initiating additional services.
- **Root Cause RC2:** By storing frequently requested data closer to the service or user, caching reduces the processing time and energy required for repeated queries, as the result can be fetched directly from the cache.
- **Root Cause RC3:** Caching decreases the frequency of repeated data transfers across the network, thereby lowering the energy consumption associated with data transmission.

Caching can be implemented at various levels, such as global caching or database-level caching [48], depending on the system's requirements and architecture. The suitability of caching as an optimization depends on the characteristics of the data exchange system. For systems where the same data is accessed

or transmitted repeatedly, caching can be highly beneficial. However, in systems with dynamic data or strict consistency requirements, caching may introduce challenges such as synchronization overhead or added latency.

4.2.3 Data Compression

Data compression, the process of storing information in a compact form [51], can significantly reduce energy consumption. This is because the energy required for pre-processing tasks, such as encoding, is in most cases negligible compared to the energy consumed during data broadcasting [52]. The primary purpose of compression is to minimize storage and bandwidth requirements for data transmission, which inherently leads to energy savings. By reducing the size of data transferred over networks or stored on disk, compression decreases the resources and energy required for these operations [52–54].

In essence, smaller data volumes require less storage, which in turn lowers the energy demands for maintaining storage systems [45]. Multiple studies, such as [47, 52–54], consistently demonstrate that data compression is an effective energy efficiency optimization, resulting in reduced energy consumption.

4.2.3.1 Relation to Data Exchange Archetypes & Root Causes

In the context of data exchange archetypes, data compression enhances efficiency by reducing the volume of data exchanged between services or components. This optimization addresses the following root cause:

- **Root Cause RC3:** Smaller data sizes result in fewer data packets being transferred, reducing the transfer time between services and, consequently, the energy consumption associated with network usage.

The suitability of compression depends on the specific characteristics of the data exchange system. In systems with high data transfer volumes, compression can significantly improve energy efficiency and performance. However, compression may introduce latency when data must frequently be decompressed, or if the compression algorithm is computationally intensive. Furthermore, systems with minimal data transfers may benefit less from compression, as the energy savings from reduced transfers may be outweighed by the energy consumed during compression and decompression. Therefore, the suitability of data compression as an optimization should be carefully evaluated based on the system's data transfer patterns and operational requirements.

4.2.4 Efficient Communication Protocols

Selecting the right communication protocol could further optimize energy efficiency in data exchange systems. Various communication protocols exist, each with distinct energy consumption levels and characteristics. The following protocols were considered:

- **HTTP (HyperText Transfer Protocol)**: A foundational request/response protocol for the Web. HTTP operates using a client-server model where a Web Client sends a request to a Web Server, which processes it and sends back a response [55, 56].
- **Advanced Message Queueing Protocol (AMQP)**: A lightweight, standardized messaging protocol designed for secure, reliable, and interoperable communication. It supports publish/subscribe models, queuing, routing, and transactions. AMQP is widely used in business messaging systems requiring robust message delivery guarantees [55–57].
- **Message Queue Telemetry Transport (MQTT)**: A lightweight, low-overhead protocol based on the publish/subscribe model. MQTT is designed for lightweight communications in constrained or unstable networks, making it ideal for applications requiring minimal power and bandwidth and machine-to-machine (M2M) communication [55–57].
- **Constrained Application Protocol (CoAP)**: Modeled after HTTP but operating over UDP instead of TCP to minimize overhead. CoAP supports request/response and publish/subscribe models, offering confirmable and non-confirmable messaging for varying reliability needs. It is efficient for constrained environments due to its low bandwidth and energy consumption [56, 57].
- **Extensible Messaging and Presence Protocol (XMPP)**: An XML-based open standard for real-time messaging (i.e. instant messaging). XMPP supports both publish/subscribe and client/server communication models and is widely used for instant messaging applications [57].

Based on research, the energy efficiency of these protocols is ranked as follows [55–57]:

1. MQTT / CoAP

2. AMQP
3. HTTP / XMPP

Table 4.3 summarizes the characteristics of each protocol. Notably, there are discrepancies in the literature regarding the ranking of MQTT and CoAP. While some studies [55, 57] rank MQTT as the most energy-efficient, others [56] favor CoAP due to its slightly lower bandwidth and latency requirements. Furthermore, Naik [56] found that CoAP had the lowest bandwidth and latency, slightly lower than MQTT.

Characteristic	**MQTT**	**CoAP**	**AMQP**	**HTTP**	**XMPP**
Energy Consumption	Lower	Lower	Lower / Medium	Higher	Higher
Resource Requirements	Lower	Lower	Lower / Medium	Higher	Higher
Bandwidth Usage	Lower	Lower	Medium / Higher	Higher	Lower
Latency	Lower	Lower	Medium	Higher	Lower
Security	Medium	Medium	Higher	HTTP: Lower HTTPS: Higher	Higher
Reliability	Higher	Medium	Medium	Lower	Higher
Interoperability	Lower	Medium	Medium	Higher	Higher

Table 4.3: Comparison of Communication Protocols Based on Characteristics

In Table 4.3, interoperability refers to the ability of different systems, applications, or components to communicate and work together seamlessly, even if they are built using different technologies, platforms, or programming languages. It is particularly important in systems that must integrate diverse services or third-party components. Reliability describes the ability of a communication protocol to ensure that data is transmitted, received, and processed accurately and consistently, even in the face of network disruptions, message loss, or system failures; reliable protocols minimize the risk of data loss or duplication. Bandwidth usage indicates the amount of data that can be transmitted over a network within a given period; protocols that require lower bandwidth are generally more energy-efficient and better suited for constrained or low-resource environments. Latency refers to the time delay between the initiation of a request and the receipt of a corresponding response in a communication system, where lower latency results in faster and more responsive interactions between communicating entities. Finally, resource requirements encompass the computational and memory demands that a protocol places on a device or system; protocols with lower resource requirements typically consume less energy and are preferable for devices operating under power, memory, or processing constraints [55–57].

4.2.4.1 Data Transfer Protocols

Data transfer protocols, such as gRPC and REST, operate at a different level of communication protocols and play a critical role in data exchange efficiency. Kamiński *et al.* [58] conducted a comparative analysis of gRPC, REST, GraphQL, and WebSockets, finding that gRPC is the fastest protocol for transferring data between clients and servers. However, they also observed that performance is associated with memory usage; while gRPC demonstrated the highest speed, it also consumed the most memory. In contrast, WebSockets had lower performance but required the least memory for data transfer.

In another study, Chamas *et al.* [59] compared REST, SOAP, Socket, and gRPC, focusing on energy consumption. REST emerged as the most energy-efficient protocol in most cases, whereas gRPC consumed the least energy only in a small subset of the study's scenarios. REST was followed by Socket in terms of energy efficiency, while SOAP and gRPC were found to be the least energy-efficient. The higher energy consumption of gRPC could be attributed to its greater memory usage, as noted by Kamiński *et al.*, which likely increases the overall energy consumption.

4.2.4.2 Relation to Data Exchange Archetypes & Root Causes

The choice of communication protocol plays a critical role in determining the performance, energy consumption, and overall efficiency of data transfer between system components. In distributed systems like DYNAMOS, two types of communication are particularly relevant: external communication (e.g., interactions with data analysts/users) and internal communication between the system's components or

services. Selecting an energy-efficient communication protocol can reduce energy consumption in both scenarios, and leveraging an efficient data transfer protocol can further optimize energy usage.

Communication protocols directly address the following root cause of energy consumption:

- **Root cause RC3:** Energy efficient protocols, such as MQTT and REST, minimize data size and memory usage, thereby reducing the energy consumed during data transfers.

The suitability of a communication protocol depends on the specific requirements of the data exchange system. For example, CoAP and MQTT are ideal for constrained environments that prioritize low energy consumption, whereas AMQP and gRPC are better suited for secure or high-performance systems. Each protocol has unique strengths and limitations, making them appropriate for different use cases. Therefore, selecting the most suitable protocol requires careful consideration of the system's design and operational needs.

4.2.5 Energy Efficient Programming Language

The choice of programming language can significantly influence the overall energy consumption of software, as illustrated in Figures 4.1 and 4.2. Selecting a programming language that aligns with the specific requirements of the software being developed—such as the data exchange archetypes examined in this thesis—can substantially impact energy efficiency [18, 60–63].

Total					
	Energy		Time		Mb
(c) C	1.00	(c) C	1.00	(c) Pascal	1.00
(c) Rust	1.03	(c) Rust	1.04	(c) Go	1.05
(c) C++	1.34	(c) C++	1.56	(c) C	1.17
(c) Ada	1.70	(c) Ada	1.85	(c) Fortran	1.24
(v) Java	1.98	(v) Java	1.89	(c) C++	1.34
(c) Pascal	2.14	(c) Chapel	2.14	(c) Ada	1.47
(c) Chapel	2.18	(c) Go	2.83	(c) Rust	1.54
(v) Lisp	2.27	(c) Pascal	3.02	(v) Lisp	1.92
(c) Ocaml	2.40	(c) Ocaml	3.09	(c) Haskell	2.45
(c) Fortran	2.52	(v) C#	3.14	(i) PHP	2.57
(c) Swift	2.79	(v) Lisp	3.40	(c) Swift	2.71
(c) Haskell	3.10	(c) Haskell	3.55	(i) Python	2.80
(v) C#	3.14	(c) Swift	4.20	(c) Ocaml	2.82
(c) Go	3.23	(c) Fortran	4.20	(v) C#	2.85
(i) Dart	3.83	(v) F#	6.30	(i) Hack	3.34
(v) F#	4.13	(i) JavaScript	6.52	(v) Racket	3.52
(i) JavaScript	4.45	(i) Dart	6.67	(i) Ruby	3.97
(v) Racket	7.91	(v) Racket	11.27	(c) Chapel	4.00
(i) TypeScript	21.50	(i) Hack	26.99	(v) F#	4.25
(i) Hack	24.02	(i) PHP	27.64	(i) JavaScript	4.59
(i) PHP	29.30	(v) Erlang	36.71	(i) TypeScript	4.69
(v) Erlang	42.23	(i) Jruby	43.44	(v) Java	6.01
(i) Lua	45.98	(i) TypeScript	46.20	(i) Perl	6.62
(i) Jruby	46.54	(i) Ruby	59.34	(i) Lua	6.72
(i) Ruby	69.91	(i) Perl	65.79	(v) Erlang	7.20
(i) Python	75.88	(i) Python	71.90	(i) Dart	8.64
(i) Perl	79.58	(i) Lua	82.91	(i) Jruby	19.84

Figure 4.1: Programming Languages Energy Consumption, Time and Memory [60]

Rosetta Code Global Ranking	
Position	Language
1	C
2	Pascal
3	Ada
4	Rust
5	C++, Fortran
6	Chapel
7	OCaml, Go
8	Lisp
9	Haskell, JavaScript
10	Java
11	PHP
12	Lua, Ruby
13	Perl
14	Dart, Racket, Erlang
15	Python

Figure 4.2: Programming Language Energy Consumption Ranking [61]

Notably, when comparing the findings of Pereira *et al.* [60] and Pereira *et al.* [61], slight differences in results are observed. However, both studies consistently identify C as the most energy-efficient programming language, while Python and Perl are generally considered the most energy-intensive. Go is positioned somewhere in the middle of the rankings in both studies.

4.2.5.1 Relation to Data Exchange Archetypes & Root Causes

The choice of programming language directly impacts the performance and energy consumption of software. Since the data exchange archetypes require software development, selecting an optimal programming language can significantly enhance the energy efficiency of these systems. The programming language choice addresses the following root causes:

- **Root Cause RC1:** Reducing startup time can be achieved by selecting programming languages with fast compilation, low runtime overhead, and efficient execution speeds. For example, languages such as C and Rust offer significantly higher performance compared to other alternatives, making them well-suited for minimizing initialization delays.
- **Root Cause RC2:** Efficient programming languages optimize processing time by enhancing query execution and resource utilization.
- **Root Cause RC3:** Energy-efficient languages reduce the computational overhead associated with data processing and transmission, thereby lowering energy consumption during data transfers.

The selection of an energy-efficient programming language should be guided by the specific requirements of the data exchange archetypes. For performance-critical tasks, languages such as C or Rust may be ideal due to their high efficiency and minimal runtime overhead. Conversely, Go might be a suitable choice for services where rapid development and moderate efficiency are key priorities. Tailoring the programming language choice to the system's needs can significantly reduce startup time, energy usage, and overall service latency, aligning with the design goals of the data exchange system.

4.2.6 Energy Code Smell Refactoring

Energy code smells are implementation choices at *source code level* that make sub-optimal usage of the underlying hardware resources. As a result, they provoke a higher energy consumption and decrease the energy efficiency of software execution [14, 64]. Imran *et al.* [65] demonstrate that a strategic approach to code smell refactoring can significantly reduce energy consumption. Their findings show that, on average, it can achieve up to a 13.1% reduction in energy consumption per workload and a 5.1% decrease in carbon emissions for specific applications, resulting in a significant improvement in energy efficiency. These findings show the potential of strategic refactoring to significantly enhance the energy efficiency of software applications. Examples of energy code smells include Condense Boolean Logic Statements

[66] and Clean Up Useless Code and Data [14]. Additional studies investigating code smells and general code-level energy optimizations, and their impacts on energy consumption include [64, 67–78].

4.2.6.1 Automatic Refactoring of Energy code smells

Although improvements in energy efficiency may interest developers, it is unlikely that all of them have the time to consider the energy efficiency of their code [79]. Therefore, refactoring these energy code smells should ideally be performed automatically. Connolly Bree and Cinnéide [79] outline two approaches for achieving automatic refactoring:

1. **Aggressive Refactoring**: This approach integrates an automated refactoring tool into the toolchain (see Figure 4.3), relieving developers from addressing energy concerns. When code is pushed to the version control system for review or testing, the tool automatically refactors it to enhance energy efficiency before release. Since developers do not see the refactored version, the tool can focus solely on optimizing energy use without considering traditional code quality metrics such as maintainability and reliability.
2. **Multi-Objective Refactoring**: This approach optimizes energy use while maintaining or improving code quality metrics. Unlike aggressive refactoring, it preserves code quality and does not require pre-compilation, as there is minimal impact on code quality. Instead, refactoring is performed incrementally during development, enabling developers to create energy-efficient software without compromising coding standards.

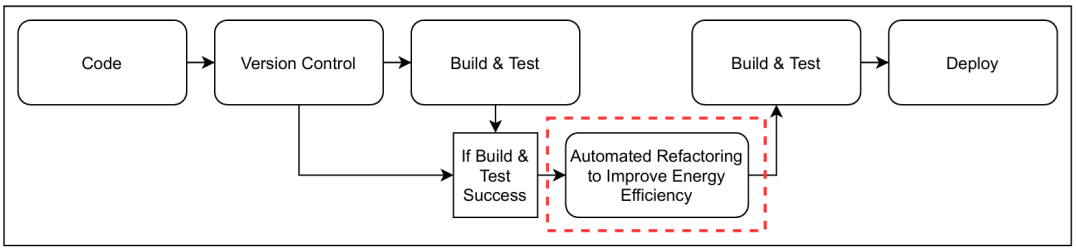


Figure 4.3: Potential Toolchain Structure for Automatic Refactoring [79]

Several automated code analysis tools, such as SonarQube, can detect software issues related to quality metrics like maintainability, reliability, and security [80]. However, during the time period of this thesis, no standard tool or technique was found for automatically refactoring energy code smells in general applications, or specifically for systems like DYNAMOS. Some tools exist for Android applications, mostly written in Java, including EcoAndroid [81], EARMO [82], Leafactor [83], and a tool proposed by Iannone *et al.* [84], but these are not applicable to general-purpose software or DYNAMOS.

So, while Connolly Bree and Cinnéide [79] proposed a potential approach to incorporate automatic refactoring of energy code smells into the software development lifecycle, there are currently few automated tools available for this purpose.

4.2.6.2 Relation to Data Exchange Archetypes & Root Causes

Refactoring energy code smells is highly relevant to data exchange systems like DYNAMOS, which depend on optimized code to reduce resource consumption. Addressing energy code smells directly targets the following root causes of energy consumption:

- **Root Cause RC1:** Eliminating unnecessary code and optimizing initialization processes can reduce startup time, enabling services to initialize more quickly and efficiently.
- **Root Cause RC2:** Improving code efficiency minimizes processing time by reducing computational overhead, leading to faster execution and lower energy consumption.

4.2.7 Minimizing Number of Data Transfers

In microservice-based architectures like DYNAMOS, each internal data transfer—where data is passed from one service to another—adds to the overall transferring time and energy consumption. As identified in the root cause analysis of DYNAMOS (Root Cause RC3), frequent data transfers can significantly

increase latency and energy usage. Reducing the number of data transfers can substantially decrease total latency and, consequently, energy consumption.

4.2.7.1 Relation to Data Exchange Archetypes & Root Causes

Minimizing data transfers is a key optimization for enhancing the energy efficiency and performance of data exchange systems like DYNAMOS. This optimization directly addresses the following root cause:

- **Root Cause RC3:** Reducing the number of data transfers minimizes transferring time, thereby decreasing the energy consumption associated with data transmission.

This optimization is particularly impactful in microservice-based systems, where frequent inter-service communication can result in significant energy usage. Consolidating data transfers ensures more efficient resource utilization, improving both performance and energy efficiency across the system.

4.2.8 Persistent Jobs instead of Ephemeral Jobs

This optimization targets data exchange archetypes in applications built using a microservice architecture. As described by Stutterheim *et al.* [9], DYNAMOS currently employs ephemeral jobs, where microservices are created on demand, executed, and then removed. This approach increases execution times due to the need to initialize new microservices for every request, contributing to Root Cause RC1. Transitioning to persistent jobs—microservices that run continuously within the cluster—can eliminate the startup time associated with creating new services for each request, potentially reducing energy consumption significantly.

4.2.8.1 Relation to Data Exchange Archetypes & Root Causes

Switching from ephemeral jobs to persistent jobs has direct implications for enhancing energy efficiency and reducing latency in microservice-based data exchange systems like DYNAMOS. This optimization directly addresses the following root cause:

- **Root Cause RC1:** Persistent jobs eliminate startup time, as services are already running and do not require initialization for each request.

Persistent jobs are particularly advantageous for systems with recurring or high-throughput data requests, as they reduce redundant initialization processes. However, this approach may increase resource usage during idle periods and introduce potential security risks [9]. The suitability of a persistent architecture depends on the operational characteristics of the data steward. For example, a data steward responsible for a limited number of frequently queried datasets and a small set of microservices would benefit significantly from adopting a persistent architecture. In contrast, a steward who manages diverse and infrequent or sporadic requests may be better served by maintaining an ephemeral job structure, which conserves resources by only instantiating services when needed [5].

4.2.9 Query Execution Optimization

Optimizing query execution has shown to improve energy efficiency in several studies, such as [45, 85]. Examples of this optimization include using efficient database queries, such as avoiding unnecessary sorting operations, to minimize computational overhead [85]. In systems not involving databases (e.g., queries on CSV files), query execution optimization could focus on reducing execution time and streamlining the retrieval process.

4.2.9.1 Relation to Data Exchange Archetypes & Root Causes

Query execution optimization is crucial for improving the energy efficiency of data exchange systems like DYNAMOS, where frequent data retrieval, processing, or analysis occurs. This optimization addresses the following root cause:

- **Root Cause RC2:** Reducing query complexity and execution time directly minimizes processing time, which can significantly decrease the energy consumed during data retrieval and analysis operations.

By optimizing query execution, computational overhead in data exchange systems can be reduced, making the system more energy-efficient while maintaining or improving performance. For DYNAMOS,

implementing such optimizations in database queries or other data retrieval processes (e.g., CSV file processing) has the potential to substantially lower energy consumption while enhancing overall system performance.

4.3 Selected Optimizations for DYNAMOS

This section details the optimizations selected for implementation in DYNAMOS. While Section 4.2 discussed how the candidate optimizations can enhance energy efficiency in data exchange systems in general, this section focuses on the specific optimizations chosen for DYNAMOS and provides justifications for both selected and non-selected options. Furthermore, the objectives of the selected optimizations for implementation in DYNAMOS are outlined. Table 4.4 presents an overview of the selected optimizations, with each optimization assigned a unique ID for ease of reference throughout this thesis.

ID	Name
O1	Caching Requests
O2	Transferred Data Compression
O3	Query Execution Optimization

Table 4.4: Selected Energy Efficiency Optimizations

4.3.1 Excluded Optimizations

Below is a detailed explanation of the excluded optimizations and the rationale for their exclusion:

- **Batch Processing**: Batch processing is unsuitable for DYNAMOS due to its architecture, which is designed to handle individual, dynamically created microservice chains for each request [5, 9]. Incorporating batch processing would conflict with this design by introducing significant latency for data stewards, as requests would need to be grouped and processed together. This optimization is better suited for systems with high volumes of frequent data exchange requests, where batching can reduce overhead and optimize resource usage.
- **Efficient Communication Protocols**: While protocols like MQTT or REST could enhance energy efficiency, DYNAMOS uses HTTP for external communication and AMQP for internal messaging to meet specific requirements (e.g., ORCH5, ORCH6, and AGENT5) [5]. Changing these protocols would violate Requirement R4 by necessitating a redesign of core communication mechanisms. Furthermore, the implementation effort required would exceed the time constraints of this thesis. Efficient communication protocols are better suited for systems in their early development stages, where communication strategies can be selected without conflicting with established designs.
- **Energy Efficient Programming Language**: The programming languages in DYNAMOS, including Go, were chosen for their suitability, particularly for seamless integration with Kubernetes, asynchronous operation, and robust compatibility with gRPC [5]. While changing the programming language could improve energy efficiency, it would introduce significant drawbacks, such as disrupting other aspects of the system. Furthermore, refactoring the entire codebase is impractical within the time constraints of this thesis. This optimization is more applicable to systems in the design phase, where selecting an energy-efficient language can prevent the need for future refactoring.
- **Energy Code Smell Refactoring**: This optimization targets lower-level energy improvements (i.e., code-level), which are less impactful for DYNAMOS because its code primarily facilitates communication in a distributed environment rather than performing computationally complex or resource-intensive tasks. The main energy bottlenecks in DYNAMOS are related to startup and transferring times (Root Causes R1 and R3), rather than code complexity. Consequently, refactoring energy code smells would provide minimal benefits compared to other strategies, such as caching (see Section 4.3.2). This type of optimization is better suited for systems with complex processing logic or high algorithmic complexity, such as loops or sorting algorithms, where refactoring can lead to substantial reductions in energy consumption.
- **Minimizing Number of Data Transfers**: While reducing data transfers could enhance energy efficiency, implementing this optimization would require removing the sidecar pattern [5, 9]. This

would violate design requirements such as DYN4 (maintaining component replaceability) and JOB4 (supporting language-agnostic microservices) [5, 9], thereby conflicting with Requirement R4. Furthermore, removing the sidecar pattern would require significant architectural changes that are beyond the scope of this thesis. Instead, this thesis focuses on optimizing transferring time within the existing architecture (see Section 4.3.3).

- **Persistent Jobs instead of Ephemeral Jobs**: Persistent jobs could eliminate startup time (Root Cause R1) but conflict with DYNAMOS' dynamic microservice creation model. Another alternative, such as removing the ability for users to dynamically add options (e.g., the "graph" option), could reduce startup time but would compromise flexibility (Requirement OS6) and introduce potential security risks (Requirement OS7) [5]. The ephemeral job model aligns with DYNAMOS' goal of dynamically creating microservice chains per request, ensuring adaptability to diverse user needs [5, 9]. Therefore, changing this model would violate Requirement R4.

4.3.2 O1: Caching Requests

Request caching has the potential to deliver substantial energy savings in DYNAMOS by addressing all identified root causes (see Table 3.6). The primary objective of this optimization is to implement request caching within DYNAMOS. This approach tackles the root causes as follows:

- **Root Cause RC1:** Caching requests eliminates the need to create jobs for identical requests, bypassing the service startup process.
- **Root Cause RC2:** Cached requests negate the need to execute the query, effectively removing processing time entirely.
- **Root Cause RC3:** Data transfer requirements are significantly reduced, as cached responses remove the necessity of transferring data between services.

A key consideration for this optimization is ensuring it does not violate DYNAMOS' design as a trust system, where policy checks are essential. For instance, if caching allows requests to bypass policy checks by directly returning cached responses, it could compromise the system's trust-based architecture. This aspect will be discussed in detail in Section 8.3.1.

4.3.2.1 Scoping

While caching can be implemented in various forms, this optimization specifically focuses on request caching. Other types of caching, such as query caching, policy caching, or distributed caching (e.g., Kubernetes state or session caching), are beyond the scope of this optimization. For instance, caching policies in DYNAMOS is not advisable because policies are subject to change and would yield limited benefits from caching, as the policy checks would need to be updated frequently. The detailed implementation of this request caching approach will be provided in Section 5.1.

4.3.3 O2: Transferred Data Compression

Compression is another optimization that holds the potential for significant energy savings in DYNAMOS by addressing Root Cause RC3: transferring time. This is achieved by compressing the data transferred between components, which can considerably reduce the size of the data being transmitted. A smaller data size could lead to a significant reduction in the time required for data transmission, improving overall energy efficiency. In the context of DYNAMOS, a primary delay stems from the data transfers between services, as illustrated in the traces shown in Figures 3.10 and 3.11. Reducing the size of the transferred data could significantly improve performance.

4.3.3.1 Scoping

This optimization is specifically focused on compressing the data transferred between components in DYNAMOS. In other words, the result of the query transferred between the components, causing the delays illustrated in Figure 3.11. Other forms of compression, such as HTTP request and response compression, are beyond the scope of this effort. The detailed implementation steps for this optimization will be discussed in Section 5.2.

4.3.4 O3: Query Execution Optimization

The third selected optimization focuses on improving query execution, primarily targeting Root Cause RC2: processing time. By optimizing how queries are executed, this approach aims to significantly reduce the time required to retrieve data. Consequently, this optimization seeks to lower the energy consumed during query execution.

4.3.4.1 Potential Scope and Setup

Due to time constraints and a focus on other topics for this thesis, such as deployment in FABRIC, this optimization is not implemented as part of this thesis. However, we did reason about a scope and setup of this implementation. This scope and setup is explained below.

While numerous techniques exist for optimizing query execution, this optimization in DYNAMOS could focus on enhancing query execution without altering the existing environment or overarching techniques. Specifically, it retains the use of Python and CSV files as data sources. Broader changes, such as switching programming languages or migrating to alternative data storage solutions like databases, are not considered within the scope of this optimization as it would require significant changes to DYNAMOS. Furthermore, this effort could concentrate solely on the execution process of queries, excluding other potential improvements, such as caching query results. This could involve implementing concurrent programming and other query execution optimizations for example.

Chapter 5

Implementations in DYNAMOS

This chapter details the implementation of our modifications to DYNAMOS. It begins with the implementation of each energy optimization and concludes with an explanation of the deployment of DYNAMOS in FABRIC. The complete project codebase is publicly available on GitHub[1].

5.1 O1: Caching Requests

This section explains the implementation of Optimization O1 in DYNAMOS: caching requests.

5.1.1 Storage Type & Caching Tool

Various caching storage types and tools are available; however, this optimization considers only in-memory caching tools. Since the primary goal is to reduce the time required for requests, in-memory storage is the most suitable choice as it provides rapid access to frequently used data by storing it in memory. In-memory caches, also known as memory caches, are high-speed data storage structures that retain frequently accessed data in the principal memory of a computer [86].

Two main tools were considered for in-memory caching: Memcached[2] and Redis[3]. Memcached supports simple key-value pair storage as strings with a size limit of 1MB per value, while Redis supports additional data structures such as lists, sets, and hashes, with a size limit of 512MB per value [87]. Redis also allows key lengths up to 2GB, whereas Memcached has a 250-byte limit [88]. Overall, Redis is more versatile, offers a broader feature set, and is generally easier to install and integrate with Kubernetes, making it a more suitable choice for DYNAMOS [87–89].

In DYNAMOS, the results of queries could exceed 1MB in size, and key lengths might surpass 250 bytes due to the need to store request bodies. Consequently, Redis is selected over Memcached as the caching tool for its flexibility, larger limits, and better suitability for Kubernetes environments.

5.1.2 Implementation

Caching was implemented in DYNAMOS by storing the results of composition requests generated for each data request. A unique cache key is created based on the composition request, incorporating elements like the user, archetype, query, and options. Certain values are hashed to optimize storage and retrieval efficiency, such as the query.

The implementation involves the following steps:

1. After all policies are checked to ensure compliance, the cache is queried using the constructed cache key.
2. If a cache hit is found (i.e., a similar request has been previously processed), the system bypasses further processing. This eliminates the need for microservice chain deployment and query execution. Instead, the cached result is retrieved and returned immediately, reducing response times to mere milliseconds and significantly improving performance.
3. If no cache hit is found, the request is executed and the result is stored in the cache with a time-to-live (TTL), which automatically removes the cache entry after its TTL expires. The TTL is set

[1] https://github.com/CollinPoetoehena/EnergyEfficiency_DYNAMOS
[2] https://www.memcached.org/
[3] https://redis.io/solutions/caching/

equal to the approval request and job TTL, which is 10 minutes in DYNAMOS. This ensures the cache remains up-to-date, avoids overload from excessive requests, and automatically cleans old entries.

4. The cache is updated only when no cache hit is found, avoiding duplicates and minimizing write/update operations, which optimizes cache performance [90].

5.1.3 UNL Scenarios

The use case for the initial development of DYNAMOS [5], as well as for this thesis, is the UNL use case, explained in Section 2.1.1. Figures 5.1 and 5.2 illustrate the differences between the original and caching-enhanced archetypes used in this thesis.

Figure 5.1 outlines the updated flow for the Compute to Data scenario, which now follows these steps:

1. **Request approval** → The user requests approval from the orchestrator, listing the intended target data stewards [5].
2. **Validation request** → The orchestrator forwards the request to the policy enforcer, which verifies existing agreements and returns an access token to the data analyst [5].
3. **Composition request** → The orchestrator selects an archetype, generates a job ID, and notifies all distributed agents of their roles within the archetype [5].
4. **Accepted request** → The orchestrator ensures all targets are online, providing the data analyst with endpoints, the job ID, and the access token [5].
5. **Data request** → The data analyst sends a data request containing queries and algorithms to the specified endpoint(s) and awaits results [5].
   - 5.1 The distributed agent checks the cache using a cache key derived from the request. If a cache hit is found, the cached result is returned to the data analyst.
   - 5.2 If no cache hit is found, the microservice chain is deployed to perform the data request. Upon completion, the result is stored in the cache for future retrieval.

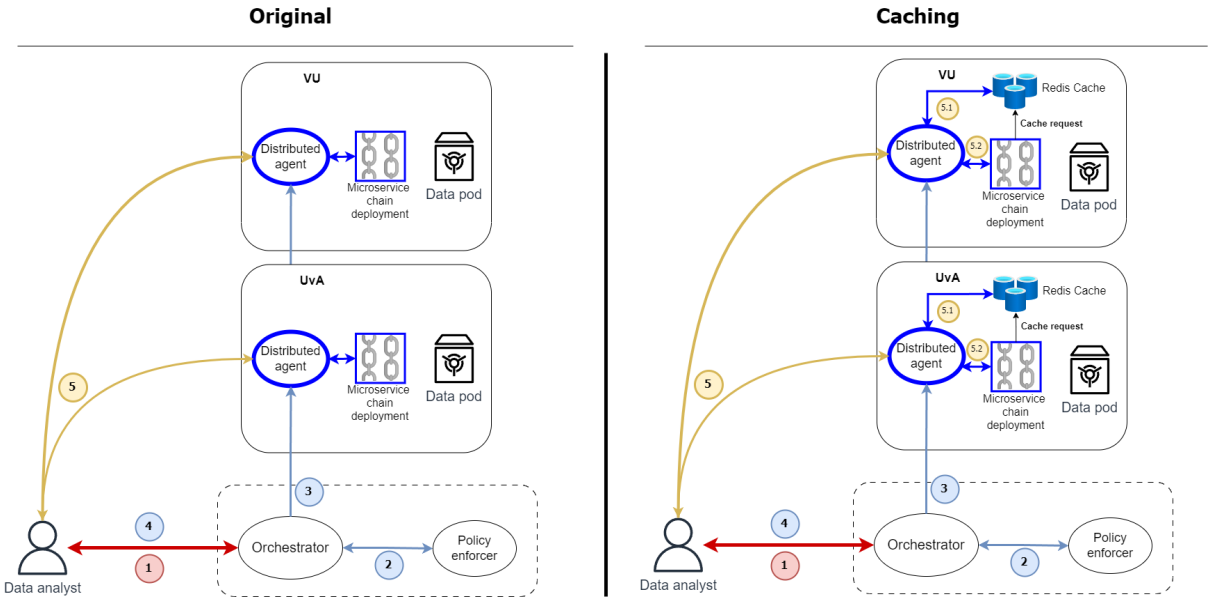


Figure 5.1: Compute to Data Scenario Original versus Caching

Figure 5.2 depicts the updated flow for the Data through TTP scenario, which extends the steps outlined above (steps 1-4 are similar):

5. **Data request** → The data analyst sends a data request containing queries and algorithms to the specified endpoint(s) and awaits results [5].
   - 5.1 The distributed agent checks the cache using a cache key derived from the request. If a cache hit is found, the cached result is returned directly to the data analyst, skipping to step 9.

   If no cache hit is found, the following steps are executed:

6. **Forward request** → The data request is forwarded to all data providers [5].
7. **Data transfer** → The data providers send the requested data to the trusted third party (TTP) [5].
8. **Process data** → The TTP further processes the data [5] and stores the result in the cache for future retrieval.
9. **Return results** → The final results are returned to the data analyst [5].

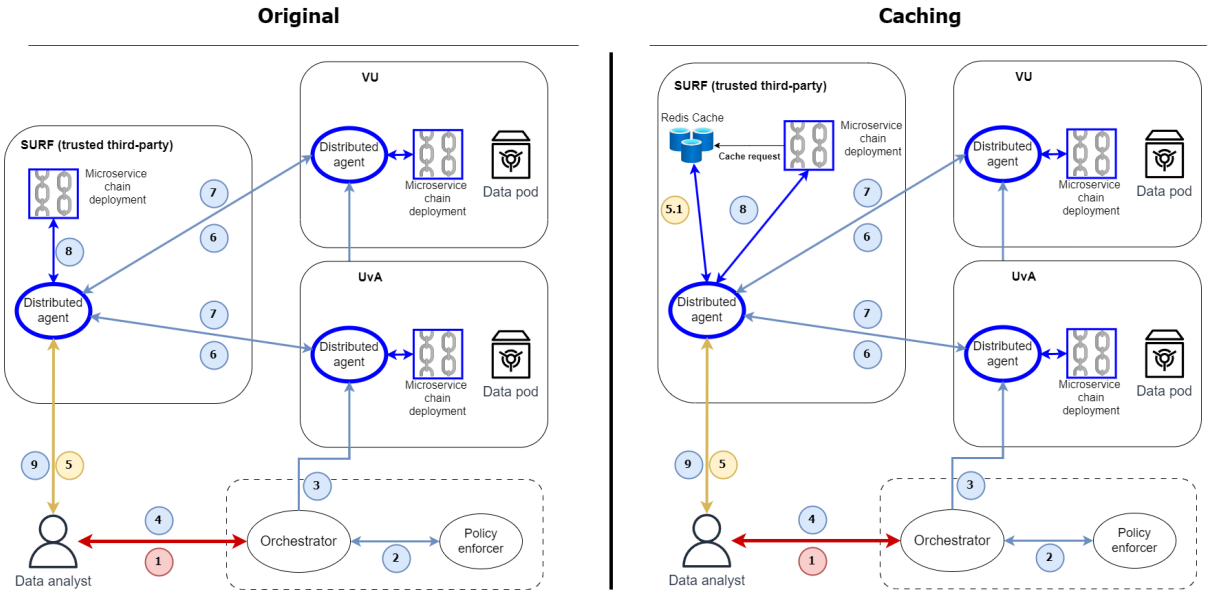


Figure 5.2: Data through TTP Scenario Original versus Caching

The primary distinction between the original and caching-enhanced archetype scenarios lies in the addition of a caching mechanism. The cache is checked before deploying the microservice chain and performing data request operations, improving efficiency and reducing redundant processing for similar requests.

5.1.4 Future Optimization Opportunities

Additional enhancements could be applied to the caching functionality, such as compressing cached data to further optimize performance. However, due to time constraints, the implementation was limited to the described approach. Avoiding further optimizations, like compression, also ensures that external factors do not influence the energy measurements, maintaining the integrity of the results to only measure the impact of this specific optimization: request caching.

5.2 O2: Transferred Data Compression

This section explains the implementation of Optimization O2 in DYNAMOS: transferred data compression.

5.2.1 Data Compression

Figure 5.3 illustrates the internal communication flow in DYNAMOS for jobs created during a data request. This flow is central to handling data requests, specifically involving the transfer and processing of query result data. The data is transferred between services within this flow, with gRPC managing the communication.

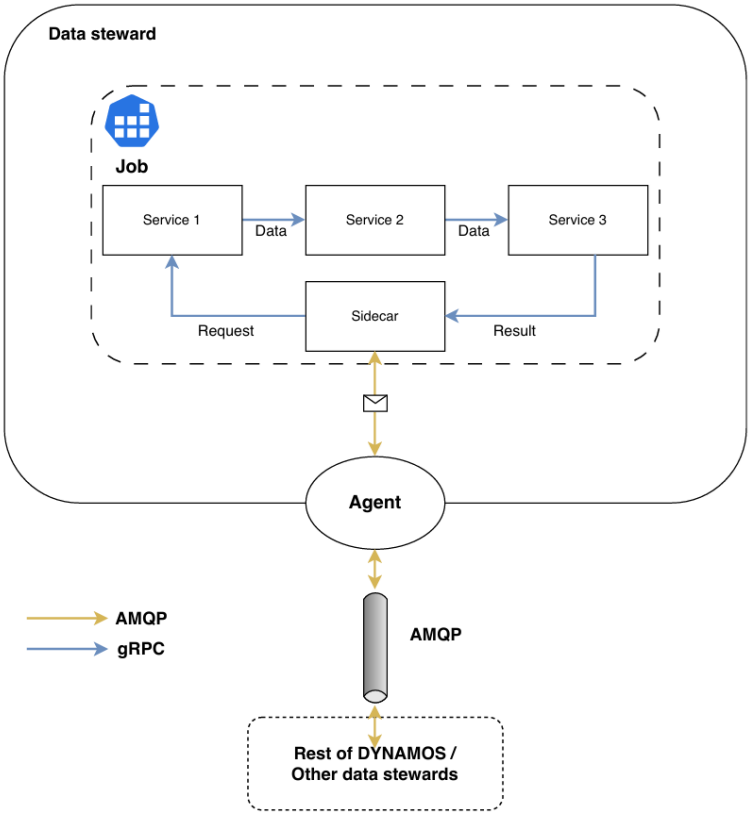


Figure 5.3: Internal DYNAMOS Jobs Communication Flow [5]

Figure 5.4 depicts another internal communication flow in DYNAMOS, focusing on the interaction between components during a data request. In this flow, data is transferred between different parties using AMQP for communication.

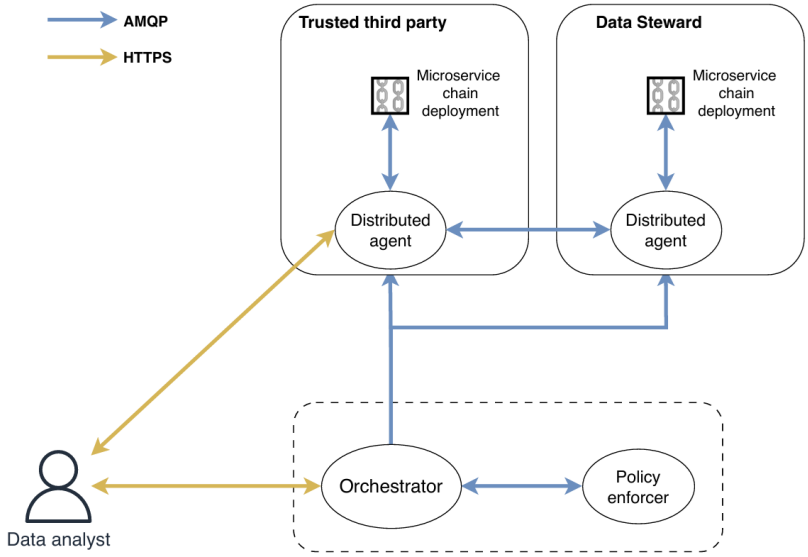


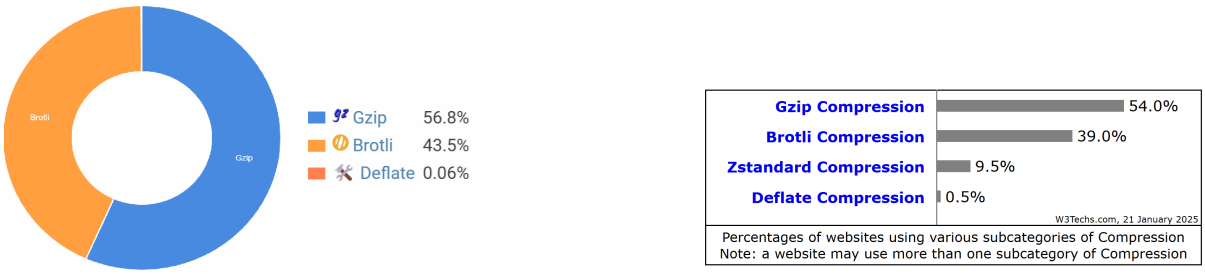
Figure 5.4: Internal DYNAMOS Components Communication Flow [5]

To implement transferred data compression within DYNAMOS effectively, it is essential to focus on the communication flows highlighted in Figures 5.3 and 5.4, as these represent the primary pathways for data transfers.

5.2.2 Compression Algorithm & Technique

Data compression techniques can broadly be categorized into two types: lossless and lossy compression. Lossless compression ensures that the decompressed data is identical to the original data, while lossy compression sacrifices some accuracy for a better compression ratio[4]. For the implementation in DYNAMOS we prioritize lossless compression because the data must remain accurate for further processing and analysis by internal DYNAMOS services and the data analyst.

Figure 5.5 highlights the most popular compression technologies, showing Gzip and Brotli as the leading techniques, with Zstandard and Deflate holding smaller market shares. Although the two figures present slightly different distributions, both indicate that Gzip ranks at the top, closely followed by Brotli. Consequently, the selection of a compression technique for DYNAMOS was narrowed down to Gzip and Brotli.was narrowed down to



(a) WMTIPS [91] **(b) W3Techs [92]**

Figure 5.5: Compression Technology Market Share Statistics

The primary distinction between Brotli and Gzip lies in their compression characteristics. Brotli achieves a higher compression ratio, offering greater size reduction, but it lacks the universal support and compatibility of Gzip. Conversely, Gzip has faster compression speeds and has been widely adopted since the 1990s, providing universal compatibility[5][6]. Studies such as [93, 94] confirm that while Brotli offers better compression ratios, Gzip outperforms Brotli in terms of speed.

The final choice for compression in DYNAMOS is Gzip. In short, Gzip is a lossless compression technique based on the DEFLATE algorithm[7], which combines LZ77 and Huffman coding [94]. Its superior compression speed aligns with the primary objective of minimizing the time required for data transfers, as identified in Section 3.3. Furthermore, Gzip's universal support and compatibility simplify integration into DYNAMOS. For example, Gzip is included in Go's standard library, whereas Brotli currently requires additional dependencies[8]. These factors make Gzip a well-balanced choice for DYNAMOS, offering an optimal trade-off between compression speed, compatibility, and performance.

5.2.3 Microservice Communication in DYNAMOS

In DYNAMOS, the messages and functions used by gRPC are defined in `.proto` files. The primary message utilized for communication between microservices is the `microServiceCommunication` message [5]. The definition of this message is presented in Listing 5.1.

```
message MicroserviceCommunication {
  string type = 1;
  string request_type = 2;
  google.protobuf.Struct data = 3;
  map<string, string> metadata = 4;
  google.protobuf.Any original_request = 5;
  RequestMetadata request_metadata = 6;
  map<string, bytes> traces = 7; // Binary or textual representation of span context
  bytes result = 8;
  repeated string routing_data = 9; // To be used for persistent jobs
}
```

Listing 5.1: Definition of MicroserviceCommunication Protobuf Message in DYNAMOS

[4] https://www.geeksforgeeks.org/what-are-data-compression-techniques/
[5] https://www.cloudways.com/blog/brotli-vs-gzip/
[6] https://wp-rocket.me/blog/brotli-vs-gzip-compression/
[7] https://en.wikipedia.org/wiki/Gzip
[8] https://pkg.go.dev/compress

This communication process corresponds to the flow depicted in Figure 5.3 for handling jobs. The first service in the flow is always the `sql-query` service, which executes the query to retrieve the data. Additional services, such as `sql-algorithm` and `sql-aggregate`, are optionally included to further process the data. These services transfer data sequentially, as shown in the communication flow in Figure 5.3, with gRPC facilitating this communication.

The final data is then transferred from the last service to the sidecar via gRPC. The sidecar subsequently sends the data to the agent using AMQP. The agent can, in turn, transfer the data to other data stewards when necessary, such as in scenarios involving a trusted third party (TTP).

5.2.3.1 Required Fields from Microservice Communication Message

By logging the contents of the `microServiceCommunication` message (see Listing 5.1) when performing data requests, we identified that each microservice modifies the `data` field. For every data request, a chain of microservices is deployed to process the data. This process follows the communication flow shown in Figure 5.3.

Initially, The `sql-query` service initiates the process by executing the query. Additional services are incorporated into the service chain based on the specific requirements of the SQL data request. For example, if the request includes an aggregate option, an `sql-aggregate` service is added to the chain. The `sql-algorithm` service is always included as the final service in the chain before the sidecar, primarily to facilitate the transfer of data to the Go-based `sidecar` (all services except `sql-query` are implemented in Go). However, the `sql-algorithm` service only performs computations if the request specifies the average algorithm or graph option. In cases where these options are not selected, the service simply sets the `result` field and forwards the data to the `sidecar`.

Each SQL service in the chain updates the `data` field of the `microServiceCommunication` message with its processed results and forwards the updated message using gRPC. The final SQL service (`sql-algorithm`) sets the `result` field in the `microServiceCommunication` message. After all SQL services in the chain have processed the data, the sidecar sends the final result to the agent via AMQP. At this point, the final output of the service chain is stored in the `result` field of the `microServiceCommunication` message. Consequently, the `data` and `result` fields contain substantial amounts of data critical to the system's operation.

The other fields in the `microServiceCommunication` message are generally smaller and therefore unlikely to benefit substantially from compression. Compressing and decompressing these smaller fields could even result in additional energy consumption due to the computational overhead, negating any potential benefits.

5.2.4 Implemented Compression and Decompression Logic

The previous section identified the fields containing large amounts of data within the microServiceCommunication message. To optimize data transfers, compression is applied to the `data` fields in the services illustrated in Figure 5.3. For the sidecar service, both the `data` and `result` fields are compressed. However, to avoid unnecessary overhead, the `result` fields of intermediate services, which are typically empty, are not compressed.

The functions responsible for data transfers between SQL services in DYNAMOS include the `SendData` function located in `go/pkg/lib/grpc_server.go` and the `SendData` function located in `go/cmd/sidecar-/grpc_server.go`. Each service in the chain uses the `SendData` function of the next client to pass along data. For example, Service 1 invokes the `SendData` function of Service 2, while Service 3 invokes the `SendData` function of the sidecar. In the example illustrated in Figure 5.3, which involves three SQL services, the sequence of transfers is as follows:

- Service 1 → Service 2 (invokes `SendData` from Service 2, located in `go/pkg/lib/grpc_server.go`)
- Service 2 → Service 3 (invokes `SendData` from Service 3, located in `go/pkg/lib/grpc_server.go`)
- Service 3 → Sidecar (invokes `SendData` from sidecar, located in `go/cmd/sidecar/grpc_server.go`)

During the initial two transfers, only the `data` field in the `microServiceCommunication` message is modified, while the `result` field remains empty. Consequently, compression is applied solely to the `data` fields within these transfer functions. Each intermediate service (e.g., Service 1 and Service 2) compresses the `data` field prior to forwarding it to the subsequent service. Upon receiving the data, the next service decompresses the `data` field for processing. Once processing is complete, the `data` field is updated by the SQL service and recompressed using the `SendData` function located in `go/pkg/lib/grpc_server.go`.

However, the final service in the chain (`sql-algorithm`) employs a different `SendData` function to handle the transmission of the completed `microServiceCommunication` message. It invokes the `SendData` from the sidecar, located in `go/cmd/sidecar/grpc_server.go`, since the sidecar is the next service in that case. This final SQL service decompresses the `data` field and processes it based on the request, such as executing an average algorithm. The processed result is stored in the `result` field of the `microServiceCommunication` message. At this stage, a separate `SendData` function, located in `go/cmd/sidecar/grpc_server.go`, sends the final `microServiceCommunication` message to the sidecar and subsequently to the agent. Before transmission, this function compresses the `result` field using the same approach as for the `data` field in earlier steps. Since the `sql-algorithm` service does not modify the `data` field, the `data` remains in its compressed state as prepared by the previous SQL service, eliminating the need for additional compression. Finally, this `SendData` function invokes `SendDataThroughAMQ`, passing the `microServiceCommunication` message, including the fully compressed `data` and `result` fields.

Once compressed, the `data` and `result` fields are used consistently throughout any potential subsequent data transfers in DYNAMOS. Since these fields are compressed at the point of transfer from the agent to the rest of the system (illustrated in Figure 5.3), subsequent transfers, such as requests involving a Trusted Third Party (TTP), also utilize compressed data. This approach takes advantage of the independent-query architecture of DYNAMOS, where each data request is self-contained, ensuring that the final processed data remains consistent and efficient after compression within the agent's job. This implementation addresses the second objective of compressing data transferred via AMQP between components, as illustrated in Figure 5.4.

Finally, before the results are returned to the requesting data analyst via HTTPS, the SQL request handler decompresses the `result` field to provide the final output.

5.2.4.1 Compression Workflow Example

The following steps outline the compression workflow in DYNAMOS for the scenario depicted in Figure 5.3, which includes an `sql-query`, `sql-aggregate` and `sql-algorithm` service:

1. **Service 1 (`sql-query`)**: Executes the query, populates the `data` field in the microServiceCommunication message, and invokes the `SendData` function of the next service to compress the `data` field before forwarding it.
2. **Service 2 (`sql-aggregate`)**: Decompresses the `data` field in the message received from the previous service, processes the data, updates the `data` field with the processed results, and uses the next services' `SendData` function to compress the updated `data` field before passing it to the next service.
3. **Service 3 (`sql-algorithm`)**: Decompresses the `data` field, processes it as required (e.g., performing an average algorithm if specified), and sets the final output in the `result` field. It then uses the `SendData` function of the sidecar to compress the `result` field and transmit the `microServiceCommunication` message to the agent via AMQP.
4. **Agent**: Processes the received `microServiceCommunication` message, which includes the compressed `data` and `result` fields. Depending on the scenario, the agent may forward the message to another data steward via AMQP or decompress the `result` field and send it to the data analyst through HTTPS.

5.2.4.2 Avoiding Compression of Small Data

To optimize the energy efficiency of compression, only the large data fields—`data` and `result`—in the `microServiceCommunication` message are targeted, while smaller fields are excluded as they do not benefit significantly from compression. Furthermore, the `result` field is only compressed if its size exceeds 100 bytes to avoid unnecessary compression and decompression overhead for small data. For instance, a result like `"avg_salary_scale_women:10.300"` represents a small dataset where the energy consumed during compression and decompression might outweigh the storage savings.

Compression for the `data` fields, however, is applied consistently, as we found that these fields typically contain large datasets that justify the overhead of compression. The `result` field, on the other hand, occasionally contains smaller outputs, such as those generated by the "average" algorithm, making selective compression essential for optimizing performance.

To keep track of the compressed and uncompressed states of the `result` field, a boolean variable, `result_compressed`, is added to the `microServiceCommunication .proto` message definition. This

addition ensures that the agent can distinguish between compressed and uncompressed `result` fields before sending it to the data analyst through HTTPS.

5.2.4.3 Small Tests of Solution

To evaluate the implementation, small and large query results were tested using Gzip. Table 5.1 presents the compression results, including the original and compressed sizes, as well as the size reduction percentage. These sizes were logged in bytes within the Go implementation. The results demonstrate that Gzip is highly effective for compressing large datasets, achieving a size reduction of 87.60%. Even for smaller datasets, Gzip provides a size reduction of over 50%, illustrating its overall efficiency.

Query Type	Original Size (bytes)	Compressed Size (bytes)	Size Reduction (%)
Small Query	545	243	55.41%
Large Query	1 135 449	140 848	87.60%

Table 5.1: Compression Results for Small and Large Queries Using Gzip in DYNAMOS

5.2.5 Alternative Approach

The current implementation compresses only specific parts of the data transferred through gRPC and AMQP, focusing on the query result data. An alternative approach could involve compressing all data communicated via gRPC and AMQP. However, this approach is beyond the scope of the current implementation, as outlined in Section 4.3.3. The primary benefit of compression in DYNAMOS lies in reducing the size of the query result data, which is typically large. Compressing smaller messages, such as metadata or control messages, would likely provide negligible benefits and could even increase energy consumption due to the additional processing required for compression and decompression.

5.2.6 Future Optimization Opportunities

Further enhancements to this implementation could involve exploring alternative or more efficient compression techniques. For example, developing or evaluating optimal compression algorithms tailored to the specific characteristics of DYNAMOS could yield additional performance improvements. However, the study of optimal compression algorithms is a vast topic that could constitute an independent thesis. Due to time constraints, this thesis focuses exclusively on the current implementation. Broader optimizations or alternative compression approaches fall outside the scope of this work, ensuring that efforts could be directed toward other priorities, such as the deployment of DYNAMOS, which is detailed in Section 5.3.

5.3 FABRIC Deployment

This section describes the deployment of DYNAMOS on the FABRIC testbed—a distributed and programmable research infrastructure. FABRIC provides low-level control over compute and networking resources, making it well-suited for evaluating energy efficiency in distributed microservice systems.

After developing and testing DYNAMOS locally, we deployed it in a multi-node Kubernetes cluster on FABRIC to validate its behavior and optimizations under real-world, distributed conditions. The setup involved configuring the cluster, deploying DYNAMOS components across nodes, integrating monitoring tools, and enabling remote access via SSH tunneling.

The following subsections provide an overview of the deployment architecture, monitoring configuration, and lessons learned during this process.

5.3.1 FABRIC

FABRIC (FABRIC is Adaptive ProgrammaBle Research Infrastructure for Computer Science and Science Applications) is a programmable research infrastructure designed to support large-scale experimentation in various domains, including networking, cybersecurity, distributed computing, storage, virtual reality, 5G, machine learning, and science applications. It provides researchers with a versatile platform to explore, test, and validate novel ideas and technologies in a controlled, scalable environment [95, 96]. Key features include:

- **Distributed Infrastructure:** FABRIC consists of a network of several sites across the United States, each equipped with substantial compute and storage resources. These sites are interconnected by high-speed, dedicated optical links, facilitating seamless data transfer and collaboration [95, 96].
- **International Expansion:** The FABRIC Across Borders (FAB) initiative extends the testbed's reach to four additional nodes in Asia and Europe. This expansion fosters international collaboration and enables experiments that require a global scale [95].
- **Integration with Other Testbeds:** FABRIC connects to specialized testbeds such as 5G/IoT PAWR, NSF Clouds, and high-performance computing facilities. This integration creates a rich environment for diverse experimental activities, allowing researchers to leverage heterogeneous resources and capabilities [95, 96].
- **Advanced Capabilities:** The infrastructure supports innovative research by providing access to programmable network technologies like P4 and OpenFlow. It also integrates machine learning and artificial intelligence, enabling novel approaches to distributed and network systems control and management [95, 96].

Researchers can request and configure resources through the official FABRIC portal, where nodes, networks, and storage can be programmatically deployed. The platform enables repeatable and controlled experiments, making it ideal for research in directions such as future internet architectures, edge computing, and cyberinfrastructure design.

5.3.2 Kubernetes Deployment Tool

We compare three main tools for deploying production-grade Kubernetes clusters: Kubeadm, Kops, and Kubespray. Other tools like Kind and Minikube are excluded, as they are primarily designed for local development and testing environments, which are not used in this thesis. Instead, we deploy Kubernetes in a production-like environment using the FABRIC testbed.

5.3.2.1 Kubeadm

Kubeadm is the official, low-level Kubernetes bootstrapping tool maintained by the Kubernetes project. It initializes a minimal, production-ready cluster by setting up control plane components and issuing certificates. However, it requires manual setup of networking, load balancing, and other essential infrastructure, making it ideal for users who want fine-grained control or to build custom setups [97–99].

The main advantages of Kubeadm include its fine-grained control over cluster setup, lightweight footprint and modularity, which make it ideal for custom deployments. It is well-documented, stable, and directly supported by the Kubernetes maintainers. However, these benefits come with trade-offs: Kubeadm does not provide built-in automation support, and it requires manual configuration for networking, storage, and other infrastructure components. Consequently, while it is highly flexible, it may not be ideal for large-scale or repeatable deployments unless supplemented with external tooling [97–99].

5.3.2.2 Kubespray

Kubespray is an Ansible-based tool that automates the deployment of Kubernetes clusters using Kubeadm under the hood. It provides a declarative, repeatable setup with support for high availability, multiple OS distributions, and a variety of networking plugins. Designed for bare metal and on-prem environments, it is ideal for reproducible research and production clusters without cloud dependencies. Under the hood it uses Kubeadm [97–99].

Kubespray's strengths lie in its ability to automate cluster deployment through idempotent Ansible playbooks, ensuring consistent state across re-runs. It supports declarative YAML-based configuration, enabling version-controlled and repeatable cluster setups, and accommodates heterogeneous environments. On the other hand, Kubespray has a steeper learning curve due to its dependency on Ansible, and it requires SSH access to all target nodes. Furthermore, its push-based provisioning model is less suited for highly dynamic infrastructures where nodes change frequently [97–99].

5.3.2.3 Kops

Kops (Kubernetes Operations) is a CLI tool that simplifies Kubernetes cluster lifecycle management on cloud providers like AWS (and to a lesser extent, GCP and OpenStack). It automates provisioning

of infrastructure, DNS, networking, and upgrades, offering a full-stack Kubernetes experience in cloud-native environments. It is not suitable for bare-metal or research testbeds like FABRIC [97–99].

While Kops simplifies Kubernetes operations and supports High Availability and rolling upgrades out of the box in cloud-native contexts, it has notable limitations. It is tightly coupled to cloud APIs and assumes infrastructure abstractions that do not exist in bare-metal or testbed environments like FABRIC. It also requires additional cloud-specific configuration such as IAM roles and DNS zones. Moreover, its support for non-AWS platforms is limited, which reduces its versatility outside of commercial cloud ecosystems [97–99].

5.3.2.4 FABRIC Integration

Kops is unsuitable for FABRIC. It is designed for cloud-native environments and relies on cloud-specific APIs (e.g., AWS, GCP) for provisioning and managing resources. FABRIC, being a research testbed with bare-metal and virtual resources, does not support these APIs or managed DNS services, rendering Kops ineffective for deployment in this context.

Initially, we selected Kubespray for deploying Kubernetes on FABRIC. Kubespray supports bare-metal and (network-)isolated environments, aligning well with FABRIC's research-oriented setup. It offers fully automated, idempotent deployments using Ansible, enabling reproducible, version-controlled setups through inventory files and playbooks. Kubespray has also been used successfully in similar contexts, such as CERN [100]. However, despite its strengths, Kubespray ultimately proved less suitable for our specific setup on FABRIC compared to Kubeadm.

While Kubespray abstracts many of the manual steps involved in deploying Kubernetes, it introduces a level of automation that can be problematic in environments with custom configurations, such as FABRIC. We encountered persistent issues during deployment, especially around networking (e.g., etcd failures due to IPv6 incompatibilities and network connectivity issues), and the extensive automation made it difficult to debug and understand what was failing. Despite dedicating over a week and multiple attempts, we were unable to achieve a stable, working cluster.

For these reasons, we pivoted to using Kubeadm. Unlike Kubespray, Kubeadm provides a low-level, transparent approach to Kubernetes cluster setup. It is directly maintained by the Kubernetes project, offering comprehensive documentation and broader community support. Though it demands more manual work—including scripting and configuring networking, certificates, and runtime environments—it gave us the control and visibility needed to incrementally debug and fix issues specific to FABRIC's VMs and networking model.

5.3.2.5 Reflection on Tooling Choice

Through practical experimentation, we observed clear trade-offs between Kubespray and Kubeadm. Kubespray provides high levels of automation via Ansible, significantly simplifying the process of deploying full Kubernetes clusters. However, this same automation makes it less flexible and harder to customize, particularly when something goes wrong. Debugging issues proved difficult, as the automated layers abstracted away the underlying processes. Furthermore, documentation and community support for Kubespray are more limited compared to official Kubernetes tools. While Kubespray performs well in standardized, cloud-like environments, it struggled to handle the non-uniform configurations and system diversity characteristics of FABRIC automatically.

Kubeadm, on the other hand, allows for fine-grained, step-by-step control over the entire setup process. It is backed by extensive, well-maintained documentation for each component, including Kubeadm itself, Docker, and Calico. This enabled a systematic and decomposed approach to problem-solving. We adopted a bottom-up strategy, starting with a single-node cluster to validate the setup and gradually scaling to a full multi-node deployment. This incremental process helped manage complexity and ensured that each step was well-understood before progressing further. In contrast to Kubespray, Kubeadm proved to be much better suited for the flexible, variable nature of FABRIC virtual machines.

Ultimately, switching to Kubeadm aligned better with our goals of achieving a deep understanding of the Kubernetes deployment process and maintaining full control over the cluster architecture on a non-standard infrastructure like FABRIC.

5.3.3 Kubernetes Cluster Setup

For this deployment, we utilize the FABRIC testbed, which provides a flexible environment with customizable virtual machines and network configurations (see Section 5.3.1). Our setup consists of a single

Kubernetes cluster deployed using kubeadm (see Section 5.3.2), comprising one control plane node and worker nodes. The control plane node is responsible for maintaining cluster state, managing workloads, and orchestrating container deployments, while the worker nodes are tasked with executing application containers and performing data exchange and computation.

In our case, a single-cluster architecture is appropriate and preferable. It avoids the overhead and complexity of managing multiple clusters while still providing the performance and scalability needed for our experiments. The single cluster handles all ingress, scheduling, and workload execution, with worker nodes processing the distributed components of the DYNAMOS system. Section 5.3.4 will elaborate on this architecture in the context of DYNAMOS specifically.

A multi-cluster Kubernetes deployment was not deemed necessary, as it would introduce additional complexity without offering clear advantages for this use case. Multi-cluster setups are typically beneficial in scenarios where strict isolation is needed between environments (e.g., development, staging, production), or where regulatory and security concerns demand separation of workloads across teams or tenants. Other justifications for multi-cluster architectures include geographical distribution to reduce latency and enhance availability and scalability beyond the capacity limits of a single cluster, or robust disaster recovery mechanisms [101, 102]. However, since none of these requirements apply to our setting within FABRIC, a single-cluster design remains the most practical and effective choice.

Similarly, deploying a high availability (HA) Kubernetes control plane with multiple control plane nodes is not required for this thesis. HA setups improve fault tolerance and are essential in production-grade environments where uptime is critical[9]. However, they add significant complexity in terms of load balancing, certificate management, and node synchronization. Given the experimental nature of this setup and the fact that FABRIC resources can be reprovisioned quickly, a single control plane node offers sufficient stability and simplicity for our goals. The additional overhead of an HA configuration is not justified in this context and adds unnecessary complexity.

5.3.3.1 Accessing VMs on FABRIC

Access to the virtual machines (VMs) provisioned in FABRIC is established via a bastion host, also known as a jump or hop node[10]. This intermediate host provides SSH access to internal VM nodes that are otherwise not directly accessible from the public internet.

SSH access to the nodes can be used to manage the Kubernetes environment, particularly during deployment, troubleshooting, and operational monitoring. From the control plane node, administrators can run kubectl commands to check cluster status, observe pod health, deploy resources, and interact with cluster components. SSH is also essential for configuring system-level dependencies such as kernel modules, firewall rules, container runtimes, and network settings. Furthermore, in failure scenarios or low-level debugging, direct access via SSH enables inspection of logs, restarting of services, or manual configuration adjustments. This hands-on access complements Kubernetes' declarative management model and is particularly valuable in non-cloud, research-driven environments like FABRIC. In our case, it was instrumental in diagnosing and resolving several configuration and connectivity issues, ultimately allowing us to progress toward a stable and functional cluster setup.

5.3.3.2 Network Setup in FABRIC

Each VM in FABRIC is assigned a management interface with either an IPv4 or IPv6 address, depending on the site[11]. Initially, we attempted to use IPv6, but encountered numerous compatibility issues, particularly with Kubespray and the etcd cluster formation process. These issues were linked to parsing IPv6 addresses and other format incompatibilities. Consequently, we opted to add an additional network (described below) to be able to use IPv4, simplifying the setup and enhancing compatibility.

To enable inter-node communication, we configured a Layer 3 (L3)[12] internal network within FABRIC using IPv4. This internal network is isolated from the public internet, ensuring secure communication between the nodes via their IPv4 addresses[13]. These addresses are then used in the Kubeadm configuration to join the cluster and enable node-to-node communication.

[9] https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/high-availability/
[10] https://learn.fabric-testbed.net/knowledge-base/logging-into-fabric-vms/
[11] https://learn.fabric-testbed.net/knowledge-base/network-interfaces-in-fabric-vms/
[12] https://en.wikipedia.org/wiki/OSI_model
[13] https://learn.fabric-testbed.net/knowledge-base/network-services-in-fabric/

5.3.3.3 Container Network Interface (CNI)-based Pod Network add-on

Kubernetes requires a Container Network Interface (CNI) add-on for enabling pod-to-pod communication across the cluster[14].

CNI plugins vary significantly in architecture, features, performance, and resource usage. In our evaluation and literature review, the following tools emerged as the most relevant:

- **Flannel** is a lightweight, Layer 3 solution that uses VXLAN for encapsulation. It is simple to deploy and resource-efficient, making it suitable for basic networking scenarios without the need for advanced security or policy enforcement. However, it does not support network policies or encryption natively [103–105].
- **Calico**, in contrast, is a more complex Layer 3 CNI that supports network policies and can operate in multiple modes (pure IP, IP-in-IP, and VXLAN). It offers strong performance under large MTU sizes, particularly in its pure IP mode, and is well-suited for environments requiring fine-grained access control. However, its complexity and reliance on multiple internal services can lead to compatibility issues in custom setups like FABRIC [103–105].
- **Canal** is a hybrid solution combining Flannel for networking with Calico for policy enforcement. It provides a compromise between the simplicity of Flannel and the advanced security features of Calico. While it offers improved policy support over Flannel alone, it still inherits some of Calico's complexity and is therefore slightly more demanding in terms of setup and resource overhead [103].
- **Cilium** is a modern CNI built around eBPF (extended Berkeley Packet Filter), enabling advanced observability and fine-grained security policies at the kernel level. It supports both Layer 3 and 4 network filtering, load balancing, and encryption. However, these features come at the cost of significantly higher CPU and memory usage, as shown in various performance benchmarks [103, 104].
- **Kube-Router** operates as a pure IP router using BGP for route distribution and replaces kube-proxy by implementing service proxying itself. It supports network policies, delivers excellent performance close to bare-metal throughput, and is relatively lightweight. Its BGP-based architecture makes it well-suited for large-scale or multi-tenant environments [104, 105].
- **WeaveNet** supports both Layer 2 and Layer 3 encapsulation and includes optional encryption. While relatively easy to set up, it incurs higher latency and CPU usage, especially under encryption. Cilium, built around eBPF, is highly extensible and supports advanced policy enforcement, but has comparatively high resource consumption [103–105].

According to benchmark studies in both academic and practical environments, Calico and Kube-Router generally outperform other solutions in throughput and latency for large MTU sizes, while Flannel remains the most resource-efficient and simple to configure for smaller, less security-critical clusters [103–105].

Initially, we used Calico, a powerful and widely-used CNI that also supports network policies. However, Calico caused persistent connectivity issues in our FABRIC environment, including failures to reach the Kubernetes API service subnet, despite extensive troubleshooting efforts like IP forwarding, manual routing additions, and restarting network components. After almost two weeks of debugging with Calico and hitting consistent issues like "network is unreachable" and "no route to host" errors, we switched to Flannel, a simpler and more lightweight CNI add-on. Flannel proved to be significantly more compatible with our setup. The key issue was resolved by explicitly adding the Kubernetes service subnet route on each node and ensuring IP forwarding and required kernel modules (such as `br_netfilter`) were enabled. The Flannel pod was also configured with `hostNetwork:true`, allowing it to access the host's routing table directly during bootstrap.

As of now, we believe that the issues with Calico stemmed from the complexity of its architecture, including multiple interdependent components and services that likely conflicted with some aspects of the FABRIC environment. In contrast, Flannel's minimalistic design and limited configuration overhead avoided these issues entirely. This change led to a stable and fully functional networking configuration. Pod-to-pod and pod-to-service communication tests were successful, both within and across nodes. Given that our setup does not require advanced features like network policies, Flannel's simplicity and reliability make it a more suitable choice for FABRIC than Calico.

Importantly, we do not require the advanced network policies provided by Calico or other feature-rich CNIs. Since FABRIC is already an isolated research environment, the Layer 3 network between nodes is not exposed to the public internet. Access to the internal VMs is controlled through a bastion host,

[14] https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/create-cluster-kubeadm/#pod-network

ensuring a level of network isolation and security by design. As a result, the added complexity of network policy enforcement was unnecessary for our setup, further supporting the choice of Flannel as a more practical and reliable solution.

5.3.3.4 Kubeadm Usage & Setup

We used kubeadm as our deployment tool due to its flexibility and transparency. Each node was manually prepared, including disabling swap, setting up required kernel modules, and configuring Docker as the container runtime—chosen to align with the Docker Desktop environment used in local DYNAMOS development. The cluster was initialized using `kubeadm init` on the control plane node, followed by joining the worker nodes using `kubeadm join` commands.

Flannel was installed by applying the official `kube-flannel.yml` manifest. Additional configuration steps included enabling IP forwarding, adding static routes to the Kubernetes service subnet, and ensuring correct kernel module loading. These manual steps, while requiring more effort compared to automated tools, enabled us to deeply understand each part of the system and address compatibility challenges specific to FABRIC. All files used for this setup are present in the GitHub repository.

5.3.4 DYNAMOS Deployment in Kubernetes on FABRIC

In this section, we outline how DYNAMOS is deployed in a Kubernetes cluster within the FABRIC testbed, including a description of the underlying cluster architecture, abstract DYNAMOS design, and the responsibilities of individual nodes across the infrastructure.

5.3.4.1 Kubernetes Cluster Architecture

Figure 5.6 illustrates the components involved in a typical Kubernetes cluster. The architecture of Kubernetes can be broken down into two main components: the control plane and the worker nodes[15]. The control plane runs core services such as the API server, controller manager, scheduler, and etcd key-value store (for internal cluster state), while worker nodes are responsible for executing the actual application workloads. Each worker node runs the container runtime, kubelet, and kube-proxy, enabling them to manage pods and communicate with the control plane.

A Kubernetes cluster is composed of several core components that work together to orchestrate containerized workloads in a scalable and resilient manner. At the heart of the system lies the *API server*, which serves as the central access point for all interactions with the cluster. Whether commands are issued via `kubectl` or through a programmatic client, they are routed through the API server, which exposes the Kubernetes API and handles all internal and external RESTful communication[16].

To persist the desired cluster state, Kubernetes uses *etcd*, a distributed and consistent key-value store that retains configuration data, such as node metadata, workload deployments, and service definitions[17]. This ensures reliability and high availability across the control plane.

Newly submitted pods are assigned to nodes by the *scheduler (sched)*, which evaluates factors such as current node resource utilization, affinity rules, and other placement constraints to make informed scheduling decisions[18].

Maintaining the desired state of all cluster components is the responsibility of the *controller manager (c-m)*, which oversees a set of controllers that monitor and reconcile the system's actual state with its intended configuration. Examples include the Node Controller (responsible for tracking node health) and the ServiceAccount Controller (which provisions service accounts in new namespaces)[19].

In cloud-based environments, the *cloud controller manager (c-c-m)* is often deployed to manage interactions with the underlying cloud provider. It handles tasks such as provisioning load balancers, managing storage volumes, and configuring network routes. However, this component is not used in our FABRIC-based setup, as the infrastructure is non-cloud-native and fully customizable[20].

Each worker node in the cluster runs a *kubelet*, an agent that communicates with the control plane and ensures that containers are running as defined in the pod specifications. The kubelet also performs health checks and updates the control plane on pod and node status[21].

[15] https://kubernetes.io/docs/concepts/architecture/
[16] https://kubernetes.io/docs/concepts/architecture/#kube-apiserver
[17] https://kubernetes.io/docs/concepts/architecture/#etcd
[18] https://kubernetes.io/docs/concepts/architecture/#kube-scheduler
[19] https://kubernetes.io/docs/concepts/architecture/#kube-controller-manager
[20] https://kubernetes.io/docs/concepts/architecture/#cloud-controller-manager
[21] https://kubernetes.io/docs/concepts/architecture/#kubelet

Cluster networking is handled by *kube-proxy (k-proxy)*, which runs on each node and manages network rules that route traffic to appropriate pods. By leveraging tools such as `iptables` or `IPVS`, kube-proxy enables Kubernetes' service abstraction and ensures that service requests are correctly routed to backend pods[22].

Finally, while not always shown in architectural diagrams, every node includes a *container runtime*—such as Docker, containerd, or CRI-O—which is responsible for pulling container images and starting containers based on instructions received from the kubelet[23].

Together, these components form the foundation of the Kubernetes orchestration model and enable robust, declarative management of distributed applications.

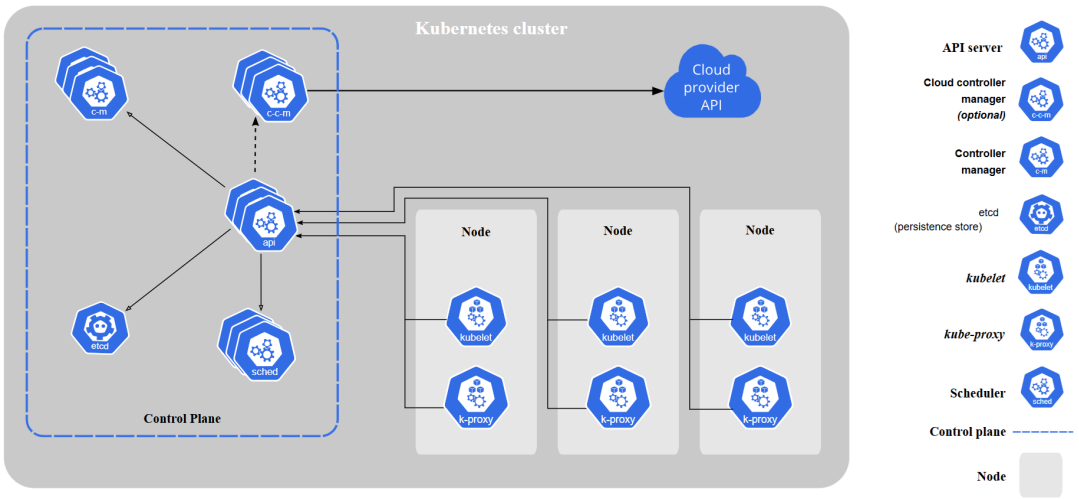


Figure 5.6: Kubernetes Components [106]

In our deployment, we configured a single Kubernetes cluster across multiple VMs on the FABRIC testbed. This included one dedicated control plane node and several worker nodes. The control plane node (node1) is solely responsible for managing and orchestrating the cluster, ensuring consistent application of configuration state, scheduling, and service discovery. To preserve isolation and reduce interference with DYNAMOS-specific components, no DYNAMOS services were deployed to the control plane node.

The separation of responsibilities between the control plane and worker nodes is essential for modularity, fault isolation, and scalability. In research environments like FABRIC—where infrastructure is provisioned explicitly—this separation also simplifies debugging and aligns well with reproducibility goals.

5.3.4.2 Abstract DYNAMOS Design

DYNAMOS, as introduced in Section 2.1, is designed around a federated data exchange model in which participants (such as organizations or research institutions) share data securely and under well-defined policy constraints. The architecture can be logically divided into three layers: the control plane, the exchange layer, and the data plane (see Figure 5.7).

[22] https://kubernetes.io/docs/concepts/architecture/#kube-proxy
[23] https://kubernetes.io/docs/concepts/architecture/#container-runtime

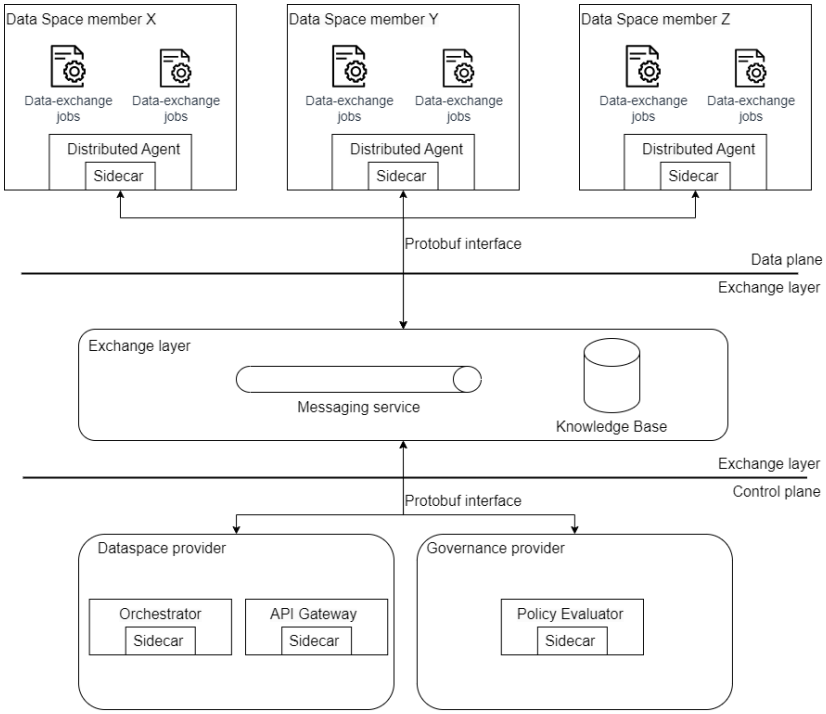


Figure 5.7: DYNAMOS Abstract Design as Obtained from Stutterheim [107]

The control plane in DYNAMOS is responsible for orchestrating workflows and evaluating access control and data usage policies. The exchange layer facilitates communication between participants and maintains metadata about data requests, often through asynchronous messaging via RabbitMQ and persistent storage via `etcd`. The data plane executes data exchange or computation jobs initiated by Distributed Agents, which are deployed at each participant's node.

Although Kubernetes internally uses `etcd` to store cluster state, DYNAMOS includes its own independent `etcd` instance for domain-specific metadata, such as job requests, agent identifiers, and policy bindings. This logical separation ensures that DYNAMOS data does not interfere with Kubernetes internals and can be versioned or queried independently.

Note: As discussed in Section 3.2.2, the original version of DYNAMOS used a unified API Gateway that combined approval and data requests into one. However, for these experiments, the system was reverted to an older, more modular architecture where the approval and data requests are separated. This change reduces computational complexity, improves fault isolation, and led to greater stability during energy measurement experiments.

5.3.4.3 DYNAMOS in Kubernetes

The deployment of DYNAMOS within our FABRIC Kubernetes environment mirrors the abstract architecture but maps specific responsibilities to separate nodes for clarity, modularity, and experimental reproducibility. Each node was provisioned with a defined role, tailored to the UNL scenario evaluated in this thesis:

- **k8s-control-plane (node1)**: This node hosts the Kubernetes control plane components and is reserved exclusively for cluster orchestration. No DYNAMOS components are deployed here to prevent interference with cluster scheduling, monitoring, or control logic. This separation supports better fault isolation and simplifies cluster management (see Figure 5.6).
- **dynamos-core (node2)**: This node serves as the operational center of the DYNAMOS system. It hosts all core components that must be globally accessible by other nodes, including the Process Orchestrator, Policy Reasoner, and API Gateway. Monitoring and observability tools such as Prometheus, Grafana, and Jaeger/Linkerd are also deployed here for convenience and central access to dashboards. However, it is important to note that monitoring itself is not restricted to this node. Most monitoring-related workloads (e.g., Prometheus Node Exporter, Kepler Exporter) are deployed as DaemonSets and run across all nodes in the cluster. These components collect node-local metrics and feed them into Prometheus, enabling a system-wide view of power consumption

and behavior. Therefore, while the visualization and aggregation layers may reside on node2, the data collection is distributed throughout the cluster. This separation supports better fault isolation and simplifies cluster management (see Figure 5.6). Furthermore, this node runs RabbitMQ for asynchronous messaging and DYNAMOS' dedicated `etcd` instance for storing metadata associated with distributed data jobs.

- **UNL, UVA, SURF nodes (nodes 3–5)**: Each of these nodes represents a distinct participant in the data exchange scenario. They run a Distributed Agent and all logic necessary to execute local data jobs or act on received data requests. This distributed deployment simulates real-world data sovereignty by assigning data responsibilities to separate VMs. By keeping one party per node, we maintain the intended design of DYNAMOS [5, 9].

Figure 5.8 illustrates this setup for the specific UNL scenario used in this thesis, including the division of responsibilities across Kubernetes worker nodes.

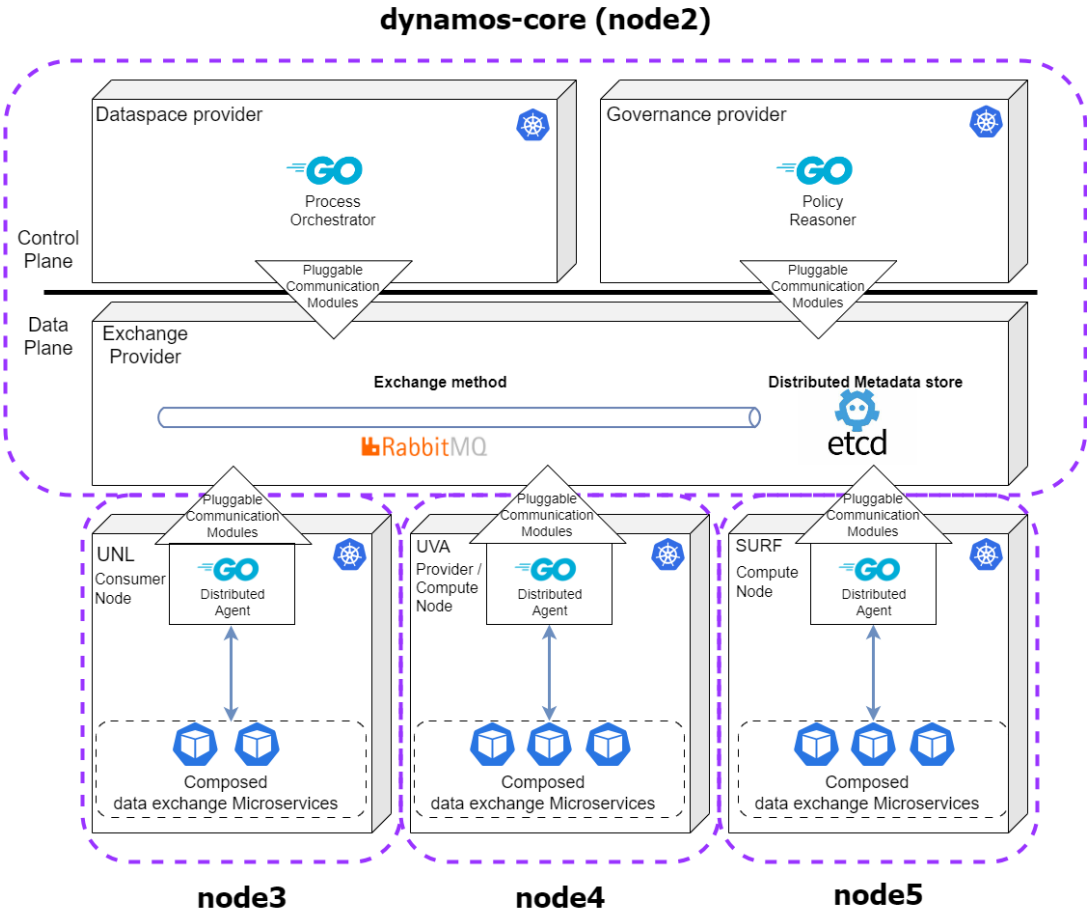


Figure 5.8: DYNAMOS Deployment Across Kubernetes Worker Nodes in FABRIC

This node structure reflects the modular philosophy at the core of DYNAMOS: each component—whether part of the central infrastructure or a Distributed Agent—can be developed, deployed, and maintained independently. This decoupling not only facilitates scalability and system evolution but also enhances observability and fault isolation. Furthermore, the deployment strategy adheres to Kubernetes best practices by clearly separating cluster orchestration (control plane) from application-level workloads (worker nodes). It also aligns with DYNAMOS' architectural vision of assigning each data exchange participant to a dedicated node, thereby preserving data sovereignty and enabling clearer traceability across federated interactions.

It is also important to distinguish between the two `etcd` instances used in this setup. The first is run internally by Kubernetes on the control plane node (node1) and is responsible for managing the Kubernetes cluster's desired state. The second is run within the DYNAMOS domain on the dynamos-core node (node2), serving as a metadata registry for the system's data exchange workflows. These

instances are entirely independent and serve different operational domains. Furthermore, the absence of node1 from the internal DYNAMOS communication network is intentional: it does not participate in the exchange and is reserved purely for Kubernetes cluster orchestration.

5.3.4.4 Energy Efficiency Optimizations in FABRIC

In the FABRIC setup, two previously evaluated optimizations—caching and compression—were included to test their effect in a distributed Kubernetes environment. These optimizations were directly ported from the local experiments and required only minimal adjustments.

For caching, a Redis instance was deployed as a Kubernetes pod on the `dynamos-core` node (node2), functioning as a shared cache accessible by all agents and orchestrators. Redis is configured to store results of prior identical requests, allowing DYNAMOS to reuse output when a computation has already been performed (see Section 5.1).

Transferred data compression was handled within the agent and sidecar services using Gzip. This optimization was enabled by default for all data responses above a certain size threshold to avoid unnecessary CPU overhead on small payloads. Since the compression mechanism is encapsulated within container logic, no Kubernetes-specific changes were required, and the optimization behaved identically in FABRIC (see Section 5.2).

The successful reuse of these optimizations across environments demonstrates the benefits of encapsulation in containerized microservices. The Kubernetes scheduler handled all service deployments consistently, confirming that DYNAMOS optimizations remain portable and effective across infrastructure backends.

5.3.4.5 Monitoring in FABRIC

To monitor power usage and energy metrics in the FABRIC testbed, we configured a `NodePort` service for both Prometheus and Grafana, which exposed the services on fixed external ports of the `dynamos-core` node. These ports are bound to the node's external IP address, making it possible to reach internal Kubernetes services from outside the cluster in a controlled manner. However, to avoid exposing these interfaces directly to the public network, we accessed them securely via SSH tunneling.

SSH (Secure Shell) is a widely used protocol that provides secure, encrypted access to remote machines over an unsecured network[24]. It supports multiple functionalities including remote shell access, file transfer, and port forwarding. One key feature of SSH is the ability to create *tunnels*, which redirect traffic from a local port on the user's machine through an encrypted connection to a specified port on a remote machine.

To access the monitoring interfaces from the local development machine, an SSH tunnel was established to the remote `dynamos-core` node. This tunnel securely forwarded a local port (e.g., `localhost:9090`) to the corresponding NodePort assigned to the Prometheus service on the FABRIC node (e.g., `30906`). The same was done for Grafana, forwarding `localhost:3000` to its assigned NodePort. This setup enables users to open URLs from their local machine such as `http://localhost:9090` in a browser and interact with the Prometheus dashboard as though it were running locally, despite it being served from within a private cluster.

In technical terms, the SSH tunnel instructs the local machine to forward traffic from a specified local port to a remote port through an encrypted SSH session. For example, the following command:

```
ssh -L 9090:localhost:30906 user@<Node-IP>
```

This tells SSH: *"Please take any request made to `localhost:9090` on my local machine, send it over the secure SSH tunnel, and on the remote FABRIC node, connect it to `localhost:30906`."* This allows for seamless, private access to Kubernetes services inside the FABRIC cluster without exposing any ports externally or requiring VPN-level access.

SSH tunneling is a common technique for securely exposing local access to services running in isolated or firewalled environments, such as testbeds, research infrastructures, or staging clusters[25]. By using this method, we were able to visualize real-time system and energy metrics collected by Kepler and other exporters during experiment runs—while maintaining strong access control and compatibility with FABRIC's research-oriented infrastructure.

[24] https://www.ssh.com/
[25] https://www.ssh.com/academy/ssh/tunneling

In addition, the official Kepler Grafana Dashboard[26] was imported into the Grafana instance. This dashboard provides a real-time overview of energy consumption (in watts) and estimated carbon emissions (in pounds of CO_2/kWh per day), broken down per pod, process, and resource category (e.g., CPU package, DRAM, disk/network, GPU).

Figure 5.9 shows a snapshot of the dashboard during the execution of one experiment. It provides valuable insights into which components of the system consume the most power and helps identify optimization opportunities.

Figure 5.9: Grafana UI Kepler Dashboard in FABRIC

The dashboard visualizes both energy usage and environmental impact using metrics collected by the Kepler exporter and stored in Prometheus. While Prometheus queries are central to the automated experiments in this thesis, this visual layer is particularly valuable during manual testing and debugging, offering immediate insight into energy behavior across the system. Based on Kepler's official Grafana dashboard JSON, the interface is designed to work seamlessly out of the box with Kepler's Prometheus metrics. It also provides flexible filtering options by namespace, pod, and node, enabling more targeted and detailed analysis.

At the top of the dashboard, three large dials display the estimated carbon emissions (in pounds of CO_2 per kWh per day) for coal, petroleum, and natural gas, based on the energy mix coefficients provided. These metrics give a quick overview of the sustainability footprint of the current workloads. Below that, the "Power Consumption" section breaks down energy usage in watts per pod or process, segmented by components like PKG (CPU cores), DRAM (memory), OTHER (e.g., disk/network), and GPU (if available). This detailed breakdown helps identify which system components and workloads are consuming the most power over time. Aggregate views, such as the total power consumption across namespaces and the kWh per day by namespace, are also presented. These visualizations provide insight into which parts of the cluster (like monitoring, kube-system, or linkerd) contribute most to the total energy footprint. This is particularly useful for researchers and developers aiming to optimize the energy efficiency of their infrastructure or benchmark the sustainability of specific services.

5.3.4.6 Energy Setup

During the local development phase using Docker Desktop, energy measurements obtained via Kepler focused exclusively on containerized workloads—such as DYNAMOS agents, orchestrators, and sidecars—alongside background processes grouped under the `system_processes` pseudo-container. The local setup, running on a single-node WSL-based VM, did not expose a separate label for `kernel_processes`. This is likely due to the limited kernel visibility in WSL or container runtime abstractions, where kernel-level tasks are not traced or differentiated. Therefore, a single-node VM, and container monitoring tools

[26] https://github.com/sustainable-computing-io/kepler/blob/main/grafana-dashboards/Kepler-Exporter.json

(like Kepler) may have not exposed `kernel_processes` as a separate label, especially if the VM kernel is abstracted or simplified, which might be the case with the WSL setup. As a result, all energy not attributed to containers may have been reported under `system_processes`.

However, when transitioning to the FABRIC environment with a real multi-node Kubernetes cluster, a new pseudo-container, `kernel_processes`, consistently appeared in the Kepler-exported metrics. This container represents energy consumption attributed to kernel-level activities such as interrupt handling, kernel threads, and system-level I/O or memory management tasks. This is likely the case because we have multiple full Ubuntu VMs (instead of a WSL VM), and Kepler has deeper access to system internals. So, each node in the FABRIC cluster runs Kepler with full access to the system's internals, allowing it to expose `kernel_processes` separately as a first-class energy consumer. This results in a more comprehensive and granular breakdown of energy usage across all nodes.

To ensure fair comparison and accurate energy measurement in the FABRIC setup, we explicitly included the `kernel_processes` container in our Prometheus queries. This inclusion ensures that kernel-level contributions—triggered indirectly by DYNAMOS actions—are not overlooked. While this introduces a slight asymmetry with the local setup, it more accurately captures the total energy impact of distributed workloads in a realistic Kubernetes deployment. This setup enhances reproducibility and transparency while acknowledging and accommodating the architectural differences between local and distributed environments. Therefore, including `kernel_processes` in the FABRIC-based experiments was deemed essential for completeness and reproducibility.

5.3.5 Overall Reflection on DYNAMOS Deployment in FABRIC

The deployment of DYNAMOS within the FABRIC Kubernetes cluster involved several technical decisions and challenges that reflect both the flexibility and complexity of working with research infrastructure.

One early design choice was to use Flannel as the Container Network Interface (CNI) for inter-node communication. While alternatives like Calico offer more advanced features, Flannel was chosen for its simplicity, reliability, and ease of integration with Kubernetes' default networking model. It performed consistently throughout the experiments and supported all inter-service communication needs without significant overhead.

Deploying the Kubernetes cluster itself proved to be the most time-intensive part of the process. Configuring the control plane and worker nodes, tuning access permissions, securing services, and validating pod networking required substantial manual setup—especially given FABRIC's fully configurable environment. This contrasts with the deployment of DYNAMOS itself in FABRIC, which was comparatively straightforward. Once the Kubernetes cluster was operational, DYNAMOS services were deployed with minimal friction due to their encapsulation in Docker containers and declarative Kubernetes manifests.

Notably, all previously implemented optimizations and monitoring mechanisms—including Redis-based caching, Gzip compression, Kepler integration, and Prometheus/Grafana dashboards—worked in FABRIC with almost no changes beyond path adjustments and environment-specific configurations. This validates a key architectural goal of DYNAMOS: to ensure deployment agnosticism and reproducibility across infrastructures.

This experience underscores a central insight from this project: while infrastructure provisioning in platforms like FABRIC requires non-trivial effort, containerized microservices and Kubernetes provide an abstraction layer that makes application-level deployment portable, consistent, and scalable. This property is particularly beneficial for energy and performance experiments, where it is important to minimize external influences introduced by environment-specific differences or inconsistencies.

Chapter 6

Experimental Setup & Design

To assess the impact of the implemented optimizations, we conducted structured experiments to measure their effect on energy consumption. This chapter details the experimental setup, methodology, and environments used to ensure a systematic and reliable evaluation of each optimization using these experiments. Importantly, the main setup and design of the experiments is similar to that of our preliminary experiments (see Section 3.2.2), but any differences or additional explanation is provided in the sections below.

6.1 Experiments Description

In this section, we outline the two experiments conducted to evaluate the impact of energy efficiency optimizations in DYNAMOS, comparing results across a controlled local environment and a distributed deployment on the FABRIC testbed.

6.1.1 Experiment 1: Local Testing

This experiment represents the initial phase of testing, conducted in a local Docker Desktop environment using a single-node Kubernetes cluster. The objective of this experiment is to establish a controlled environment to measure the impact of the implemented optimizations before extending the evaluation to a broader deployment scenario. More details about the specific experimental setup and design will be explained in the remainder of this chapter.

6.1.2 Experiment 2: FABRIC Testing

This experiment represents the second phase of testing, focusing on the deployment of DYNAMOS in a distributed environment (FABRIC). The objective is to establish a controlled yet more realistic, multi-node Kubernetes environment in order to validate the impact of the implemented optimizations. By comparing results across two different environments—local and distributed—this experiment enables a more comprehensive evaluation of energy efficiency improvements in both isolated and production-like settings. Importantly, the same experimental design and execution flow used in the local testing environment are maintained here. By maintaining a consistent experimental structure across both environments, we ensure that the results are comparable and that any observed differences are attributable to the underlying infrastructure differences, rather than inconsistencies in methodology.

6.2 Hardware

The exact same setup used in the preliminary experiments is applied to the local testing environment (see Section 3.2.2). In contrast, the FABRIC deployment introduces a few modifications. DYNAMOS is deployed on the FABRIC testbed, using a five-node Kubernetes cluster. The cluster consists of one control plane node and four dedicated worker nodes, each representing one of the roles described in Section 5.3.4. The deployment is orchestrated using `kubeadm` (see Section 5.3), with monitoring and energy observability handled via Prometheus, Grafana, and Kepler. Table 6.1 summarizes the configuration of the FABRIC System Under Test (SUT).

Parameter	Value
Software (per VM)	
Operating System	Ubuntu 24.04.1 LTS
Linux Kernel	6.8.0-49-generic
Kubernetes Version	v1.31.7
Container Runtime	Docker (docker://28.0.4)
CNI Plugin	Flannel (VXLAN)
Hardware (per VM worker node)	
VM vCPUs	8
VM RAM	16GB
VM Disk	100GB
Hardware (VM kubernetes control plane node)	
VM vCPUs	4
VM RAM	16GB
VM Disk	100GB
Cluster Configuration	
Total Nodes	5 (1 control plane, 4 worker nodes)
Kubernetes Control Plane Node Capacity	4 CPUs, 16GB memory, 110 pods
Kubernetes Worker Node Capacity	8 CPUs, 16GB memory, 110 pods
Node Roles	*Control Plane, DYNAMOS Core, UVA, VU, SURF*
Deployment Tool	Kubeadm

Table 6.1: Specifications of System Under Test (SUT) FABRIC

6.3 Measuring Energy Consumption

In this thesis, the focus is on *energy consumed* rather than *power consumed*, as the goal is to improve overall energy efficiency—minimizing total energy use over time (see Section 2.4). This focus is particularly relevant because we use VM environments, such as WSL2, where access to underlying hardware telemetry is restricted or unavailable, making power-based monitoring infeasible. As a result, we adopt a software-based approach to estimate energy consumption. Kepler was selected for this purpose because it reports energy in joules (J), aligning with our measurement goals. Its compatibility with virtualized environments, container-level granularity, and Prometheus integration made it a practical and consistent choice across both local and FABRIC deployments used in this study (see Section 3.1.2).

In the FABRIC environment, an additional consideration was introduced due to the distributed architecture of the DYNAMOS deployment. Specifically, Kepler exposes an extra pseudo-container labeled `kernel_processes`. This container appears in FABRIC's Kepler metrics, but not in local Docker Desktop setups (see Section 5.3.4). To account for this, the Prometheus query used to collect energy data in FABRIC was updated to include this label, as shown in Listing 6.1.

```
sum(increase(kepler_container_joules_total{container_name=~"kernel_processes|system_processes|
    uva|vu|surf|sql.*|policy.*|orchestrator|sidecar|rabbitmq|api-gateway"}[2m])) by (
    container_name)
```

Listing 6.1: Prometheus Energy Measurements Data Query in FABRIC

6.4 Runs Execution & Measurement

To measure energy consumption per task, we adopt the method proposed by Koedijk and Oprescu [18], in which tasks are executed repeatedly and the total energy consumption is divided by the number of runs. This averaging technique helps reduce the influence of idle energy consumption on the results. Furthermore, idle power is subtracted from the total measured energy to isolate the energy specifically

attributable to task execution. In each experiment, seven tasks are performed per run. This methodology has been validated in prior experiments and shown to produce consistent and reliable results, as detailed in Sections 3.2.2 and 3.2.3.

6.4.1 Used Task in Runs & Other Influences

The task for each run consists of a data request in DYNAMOS. While DYNAMOS supports multiple scenarios—such as integrating an aggregate service or applying average algorithms—these variations could introduce biases in the energy data. For example, caching may yield greater energy savings in scenarios involving multiple services, as jobs are deployed only once rather than for every request. In contrast, compression may not have the same benefit, as additional services increase the need for repeated compression and decompression. Testing all possible scenarios would require an impractically high number of repetitions to achieve statistical significance.

To mitigate these risks, we focus on a single scenario, as described in Section 3.2.2. This default DYNAMOS configuration, which excludes optional services like the aggregate service, minimizes the potential for bias in our data. The only variation between archetypes is the URL used for data requests, e.g., `uva` for Compute to Data and `surf` for Data Through TTP. Background processes are terminated to reduce their influence, as outlined in Section 3.2.2.

6.4.2 DYNAMOS Approval Requests

A key difference in the tasks used for these experiments is the inclusion of a request approval step before each data request, rather than approving once and reusing the approval for subsequent requests within the experiment, which was used for our preliminary experiments. While there is no internal difference between reusing an approval and requesting approval multiple times within the same archetype, there is a semantic distinction, i.e., a difference in meaning. As outlined in Section 2.1, the archetypes primarily demonstrate how data is exchanged, which is fundamentally tied to a policy-based approach. To maintain alignment with this policy-driven framework, we decided to perform the request approval step for each data request individually for these experiments.

6.5 Number of Experiment Repetitions

Cruz, a researcher with over 1600 citations[1], suggests 30 repetitions as the ideal number for reliable experimental results [38]. Following this recommendation, we repeat each experiment at least 30 times. The minimal total time of each experiment is calculated as follows:

$$\text{Number of Experiments} = 3 \text{ implementations} \times 2 \text{ archetypes} \times 30 \text{ runs} = 180$$

$$\text{Time} = 2 \text{ minutes idle} + 2 \text{ minutes active} + 0.5 \text{ minutes cool-down} = 4.5 \text{ minutes}$$

$$\text{Total Time} = \text{Number of Experiments} \times \text{Time} = \boxed{810 \text{ minutes}} = \boxed{13.5 \text{ hours}}$$

This estimate excludes the time required for redeploying components in Kubernetes and other manual tasks. So, in reality, these experiments took much longer than anticipated. If results fail normality tests or time allows, an additional 20 runs will be performed for each implementation and archetype to improve data reliability.

6.5.1 Final Experiments Execution Plan

An automated script is used to execute experiments, collect energy consumption data, and export relevant metrics, such as execution time, for analysis. Between each implementation experiment, a 5-minute rest period is applied. The experiments execution flow for each implementation is as follows:

1. Execute experiments for archetype 1.
2. Apply a short rest period.
3. Execute experiments for archetype 2.
4. Apply 5-minute rest period.

This process is repeated for all three implementation scenarios: baseline, compression and caching. Manual steps are required for redeploying DYNAMOS between implementations, but archetype switching could be automated to streamline the process.

[1] https://scholar.google.com/citations?user=O13oaH0AAAAJ&hl=en&oi=sra

6.6 Statistical Analysis

In this section, we describe the statistical methods used to validate the experimental results, detect anomalies, and interpret the impact of optimizations across both test environments.

6.6.1 Anomaly Detection

We applied an anomaly detection mechanism to filter out irregular data points and minimize the influence of anomalies, such as failed requests or sudden spikes in energy consumption. Koedijk and Oprescu [18] define anomalies (or outliers) as data points that significantly deviate from the majority of observations, and recommend the use of clustering algorithms for detecting such anomalies.

For anomaly detection, we employed the DBSCAN algorithm, as used in previous research by Koedijk and Oprescu. DBSCAN evaluates each data point by counting the number of neighboring points within a predefined distance. Points with sufficient neighbors are classified as core points, while points close to core points but lacking enough neighbors are classified as border points. Any remaining points are labeled as anomalies [18]. This approach enables us to systematically exclude anomalous energy data from our analysis.

For the configuration and threshold selection of our anomaly detection process, we initially adopted the same DBSCAN algorithm parameters as used by Koedijk and Oprescu. However, to refine the detection process, we introduced a manual verification step to optimize the threshold value in our specific implementation of the DBSCAN algorithm. This involved starting with a low threshold to detect a broader range of anomalies and then iteratively adjusting the threshold for each dataset to determine an optimal value. This manual fine-tuning was feasible due to the relatively smaller size of the energy consumption datasets of our experiments, allowing for a more precise configuration of the DBSCAN algorithm tailored specifically to our data. This approach ensured that the algorithm effectively distinguished genuine anomalies from expected variations in energy measurements.

Finally, we removed any experiment runs where a request was unsuccessful, i.e., those returning a status code other than 200. This ensures that failed executions do not skew the energy measurements.

6.6.2 Normality

Many statistical tests assume that data follows a normal distribution [18]. To determine whether our data meets this assumption, we applied the Shapiro-Wilk (W) normality test[2], which has been shown to be more powerful than other normality tests [18, 108, 109].

For our normality testing, we set the significance level (alpha) to 0.01, following the recommendation of Koedijk and Oprescu [18]. This threshold ensures that distributions close to normality can still be used for certain statistical tests. A p-value below 0.01 indicates that the data is not normally distributed.

The results of the normality test in Chapter 7 indicated that not all of our data follows a normal distribution. As a result, we selected statistical tests that do not assume normality, favoring non-parametric methods, since parametric methods require data to have a normal distribution[3].

6.6.3 Statistical Significance & Effect Size

Statistical significance determines whether observed differences or relationships in the data are likely genuine or simply due to random variation. Meanwhile, effect size quantifies the magnitude of an effect, offering insight beyond mere statistical significance [78].

6.6.3.1 Mann-Whitney U (MWU)

A widely used test for statistical significance is the Student's t-test [18] or its alternative: the Welch's t-test [78], which assume a normal distribution. However, since our data is not normally distributed, we opted for the Mann-Whitney U (MWU) test instead. MWU is a non-parametric test that compares the rankings of values in two independent groups [110]. The p-value from this test indicates whether there is a statistically significant difference between two groups [18, 78].

The Mann-Whitney U test computes two U statistics, reflecting the extent to which one group's ranks are systematically higher or lower than the other's. The smaller U value is used for significance testing[4].

[2] https://en.wikipedia.org/wiki/Shapiro-Wilk_test
[3] https://www.geeksforgeeks.org/difference-between-parametric-and-non-parametric-methods/
[4] https://en.wikipedia.org/wiki/Mann-Whitney_U_test

We applied an alpha level of 0.05, meaning a p-value below 0.05 is considered statistically significant.

6.6.3.2 Rank Biserial Correlation (RBC)

Rank biserial correlation (RBC) is a measure of effect size specifically used in conjunction with the MWU test [78, 111]. It produces a score ranging from -1 to 1, where:

- Values near 1 indicate a strong positive effect (one group ranks consistently higher).
- Values near -1 indicate a strong negative effect (one group ranks consistently lower).
- Values near 0 indicate little or no effect.

We interpret these values using the Guilford scale, as applied by Koedijk and Oprescu [112]:

- 0.0 - 0.2 → slight correlation
- 0.2 - 0.4 → low correlation
- 0.4 - 0.7 → moderate correlation
- 0.7 - 0.9 → high correlation
- 0.9 - 1.0 → very high correlation

Negative values follow the same scale, with positive correlations interpreted as improvements and negative correlations as declines in energy efficiency.

6.6.3.3 Application of Statistical Tests

To assess energy efficiency improvements, we compared the mean total energy consumption and the mean execution time per task. For each archetype, the baseline configuration (i.e., without optimizations) was evaluated against each implemented optimization. These comparisons aimed to determine whether the differences in energy consumption were statistically significant, and to quantify their practical relevance by measuring the effect size of any observed changes.

6.6.4 Correlation

A common correlation measure is the Pearson coefficient, which assumes normally distributed data [18, 113]. Since our data is not normally distributed, we instead used the Kendall Tau coefficient, a rank-based correlation measure that does not assume normality [18, 114].

We applied the same Guilford scale as in RBC to quantify our correlation results. Our goal was to determine whether a relationship exists between execution time and energy consumption. Unlike our previous comparisons that analyzed energy consumption per optimization, this analysis considered all experimental data to assess whether execution time is a reliable indicator of energy efficiency.

Chapter 7

Results

In this chapter, we present the results of the experiments conducted to evaluate the energy efficiency of the implemented optimizations in DYNAMOS. The results are structured into several key sections to provide a comprehensive and modular overview.

We begin by detailing the number of experiment repetitions and the approach to handling anomalies in the collected data. Following this, the overall experiment results are summarized, offering a high-level view of how each optimization impacted energy consumption.

Subsequent sections provide a breakdown of the results for each archetype—Compute to Data (CtD) and Data through TTP (DtTTP)—across both local and distributed environments. We then investigate the relationship between energy consumption and execution time, examining the stability and correlation patterns across experiment runs.

Finally, we include a section on additional experiments conducted to provide further insight into some of the observed behaviors. These supplementary experiments—such as the environment cluster setup comparison and archetype setup iterations—support the primary findings and help contextualize unexpected or unexplained results. Together, the results offer a robust foundation for the discussion in the following chapter.

7.1 Total Experiment Repetitions & Anomalies

This section provides an overview of the total number of experiment repetitions and the anomalies detected during data collection. Initially, the target was to conduct 30 repetitions for each experimental setup. However, after observing that the results of both the local and FABRIC experiments did not consistently follow a normal distribution, and considering the available time, we extended the number of repetitions by approximately 20 per setup to improve the reliability of our results.

The outcomes for both experiments are summarized in Table 7.1 and Table 7.2, which follow a consistent structure. Each table presents the total number of experiment runs, the number of anomalies detected, the count of invalid results due to non-200 HTTP response status codes, and the final number of valid repetitions used in the analysis.

The "Anomalies" column reflects the number of repetitions excluded due to irregular energy values—such as unusually high or low readings—identified through anomaly detection techniques.

The "Non-200 Status" column indicates how many repetitions were invalidated due to at least one run within an experiment repetition returning an HTTP status code other than 200, indicating a failed request. This value is distinct from the anomalies count, and together they represent the total number of repetitions excluded from the original dataset. For example, if nine repetitions were removed due to anomalous energy values and one due to a non-200 status code, a total of ten repetitions were excluded from the initial count.

The column "Valid Experiments" reports the number of experiment repetitions that remained after the removal of anomalies and failed requests. Although each setup was executed approximately 50 to 65 times for every implementation and archetype, the final dataset contains fewer valid entries due to this anomaly filtering process.

7.1.1 Experiment 1: Local Testing

Table 7.1 presents a summary of the total number of repetitions, anomalies detected, failed runs due to non-200 HTTP status codes, and the resulting number of valid experiments for each implementation and archetype in the local testing environment.

Compute to Data				
Implementation	**Repetitions**	**Anomalies**	**Non-200 Status**	**Valid Experiments**
Baseline	54	6	0	48
Compression	54	5	2	47
Caching	51	0	0	51
Data through TTP				
Implementation	**Repetitions**	**Anomalies**	**Non-200 Status**	**Valid Experiments**
Baseline	57	4	12	41
Compression	60	5	12	43
Caching	55	3	2	50

Table 7.1: Experiment Repetitions and Anomalies Experiment 1

The slight variation in the number of experiments across implementations is attributed to occasional instability in DYNAMOS caused by minor underlying bugs. Section 8.3.2 provides a detailed explanation of these instabilities and the mitigation strategies we applied.

These stability issues occasionally resulted in complete system crashes, requiring manual redeployment and re-execution of the affected experiment. To maintain dataset integrity, we distinguished between isolated request failures and systemic crashes. Specifically, any run that returned a status code of `400`—a pattern that consistently followed a full system crash—was excluded from the dataset *before* the anomaly detection process began. Retaining these would have unfairly skewed the results by introducing a cascade of artificially inflated errors.

In contrast, individual request timeouts (status code `408`) were retained, as they did not indicate a complete system failure but rather isolated disruptions within otherwise functional experiment runs. These were considered part of the natural variability and were included in the anomaly detection process, unless they were part of a systematic crash.

In cases where a crash occurred during an experiment, only the first failed request (typically marked by a status code `408`) was counted as a single invalid experiment repetition (as indicated in the "Non-200 Status" column). All subsequent failures resulting in status code `400`, caused by the system being in an unrecoverable state, were discarded to preserve the fairness and accuracy of the statistical analysis.

Notably, the Data through TTP archetype exhibited more frequent instability than Compute to Data, as reflected in the number of invalid results shown in the "Non-200 Status" column. This is further discussed in Section 8.3.2. To account for this, we conducted additional runs for the Data through TTP archetype, anticipating a higher anomaly rate. This is confirmed in Table 7.1, where a greater number of anomalies were detected in the Data through TTP archetype compared to Compute to Data.

7.1.2 Experiment 2: FABRIC Testing

Table 7.2 presents a summary of the total number of repetitions, anomalies detected, failed runs due to non-200 HTTP status codes, and the resulting number of valid experiments for each implementation and archetype in the FABRIC testing environment.

Compute to Data				
Implementation	**Repetitions**	**Anomalies**	**Non-200 Status**	**Valid Experiments**
Baseline	60	9	0	51
Compression	56	9	1	46
Caching	65	9	0	56
Data through TTP				
Implementation	**Repetitions**	**Anomalies**	**Non-200 Status**	**Valid Experiments**
Baseline	60	5	1	54
Compression	61	11	1	49
Caching	58	15	0	43

Table 7.2: Experiment Repetitions and Anomalies Experiment 2

As with the local experiments, the anomaly detection criteria and the handling of failed runs follow the same methodology explained in Section 7.1.1. This includes discarding repetitions affected by anomalous energy values or invalid status codes, and applying the same threshold logic when a crash occurs mid-experiment.

In contrast to the local setup, the number of repetitions for each FABRIC experiment shows slightly more variation. This is due to the unpredictable nature of DYNAMOS' stability, where occasional crashes required experiments to be restarted. In anticipation of such issues—particularly during unattended overnight experiment runs—we deliberately executed some configurations more frequently than others to ensure a sufficient number of valid results.

As reflected in Table 7.2, all implementations were repeated between 55 and 65 times. This precautionary strategy ensured that, even after filtering out anomalies and failed requests, each archetype retained a sufficient number of valid data points for meaningful analysis.

In contrast to the local testing phase, the Data through TTP archetype did not exhibit a notably higher number of anomalies in the FABRIC environment. In particular, the caching implementation showed relatively few invalid results, aligning with previous observations from Section 7.1.1 that indicated increased stability for this optimization.

7.2 Overall Results

This section provides a high-level summary of the experimental results. It synthesizes the outcomes of both local (Experiment 1) and distributed (Experiment 2) tests to evaluate the effectiveness of caching and compression optimizations in DYNAMOS.

The outcomes for both experiments are summarized in Section 7.2.1 and Section 7.2.2, which follow a consistent structure. Each section begins with a summary table that presents the mean energy consumption, standard deviation, mean execution time per task, and the p-values from the Shapiro-Wilk test used for normality assessment. This is followed by box plots that visually represent energy consumption across experiments, comparing both archetypes and all implementation variants. Lastly, each section provides a written interpretation of the results, offering observations on execution time, anomalies encountered, and the relative impact of each optimization.

Tables 7.3 and 7.4 presents an overview of the mean energy consumption and execution time for each implementation and archetype. The mean energy consumption is calculated as the average difference between idle and active energy consumption across all experiments. The execution time represents the mean time per data request (i.e., task), with each experiment consisting of seven requests (see Chapter 6). The "SD" columns represent the standard deviation for the energy and execution time mean values. Finally, the "Energy W (p)" column displays the p-values from the Shapiro-Wilk test applied to the energy consumption data, assessing whether it follows a normal distribution.

7.2.1 Experiment 1: Local Testing

Table 7.3 presents a summary of the overall energy consumption, execution time, and standard deviation values for each implementation and archetype in the local setup. The final column ("Energy W (p)")

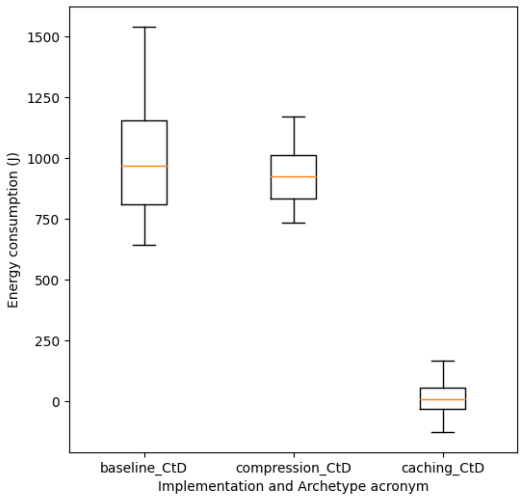
shows the p-values from the Shapiro-Wilk test for normality. Specifically, the baseline implementation for Compute to Data and the compression implementation for Data through TTP exhibit p-values below the predefined threshold (see Chapter 6), indicating a deviation from normality and justifying our choice of non-parametric statistical tests for further analysis.

It is important to note that the average execution time per task for caching may be misleading. Since caching returns results in milliseconds, the execution time is extremely low in most cases. However, every 10 minutes—corresponding to the configured time-to-live (TTL)—the cache is automatically cleared. This forces the system to reprocess a request in full rather than serve it directly from the cache. During these cache reset moments, execution times temporarily resemble those of the baseline implementation. As a result, the reported average execution time for caching is not in the millisecond range, even though most requests are returned almost instantly. For example, if the first request in an experiment run takes approximately 6 seconds due to a cache miss, while all subsequent requests are served from the cache in just a few milliseconds, the average execution time across all seven requests will still hover around one second. This effect skews the average upward and should be considered when interpreting the results.

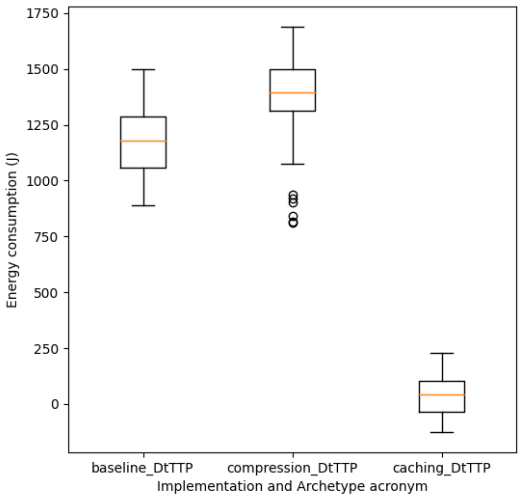
Compute to Data					
Impl.	**Energy (J)**	**SD (J)**	**Time/task (s)**	**SD (s)**	**Energy W (p)**
Baseline	1003.871	249.825	7.598	1.532	0.0042
Compression	928.958	125.386	7.324	1.105	0.0600
Caching	13.153	69.026	0.397	0.487	0.6236
Data through TTP					
Impl.	**Energy (J)**	**SD (J)**	**Time/task (s)**	**SD (s)**	**Energy W (p)**
Baseline	1174.316	172.401	8.327	0.718	0.2065
Compression	1348.404	236.661	8.592	1.053	0.0002
Caching	39.120	92.453	0.582	0.513	0.0911

Table 7.3: Energy Consumption and Execution Time Results for Implementations and Archetypes Experiment 1

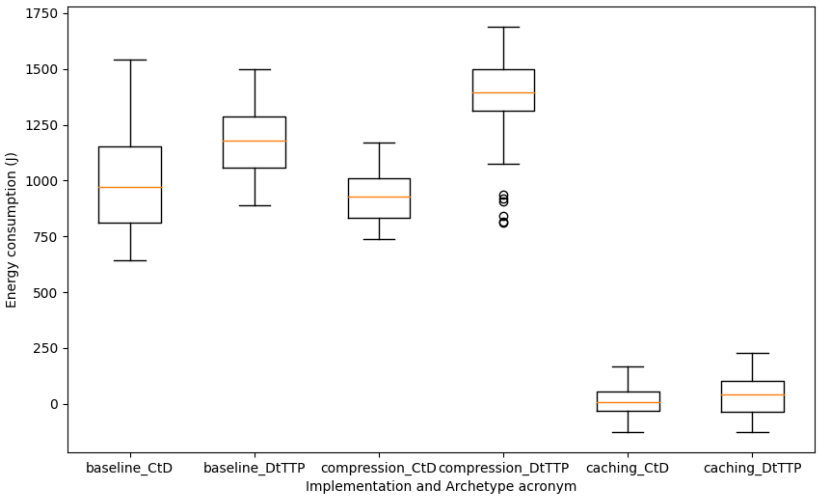
Figure 7.1 illustrates box plots for energy consumption across each archetype separately and combined. Outliers, represented as circles in the Data through TTP box plot, were not removed by our anomaly detection algorithm, as the data points considered anomalies in the box plot are approximately six data points, counting it as a cluster. These occasional lower energy reports for Data through TTP with compression are likely attributed to the instability of DYNAMOS, which is more pronounced in the Data through TTP archetype (see Section 8.3.2). To ensure transparency, these variations are retained in the results and reflected in the average values.



(a) Energy Box plot CtD



(b) Energy Box plot DtTTP



(c) Energy Box plot CtD and DtTTP

Figure 7.1: Energy Consumption Box plots for Compute to Data (CtD) and Data through TTP (DtTTP) archetypes Experiment 1

Overall, caching demonstrates the lowest energy consumption and best performance across both archetypes. Notably, in some cases, caching even results in negative energy consumption, meaning that less energy is consumed during the active period compared to the idle period. This phenomenon likely occurs because the active period energy consumption for caching is so low that it occasionally falls below the idle energy consumption. However, this does not impact the mean value, which remains a positive value. Finally, while compression exhibits slight variations from the baseline, further analysis is required to determine its statistical significance.

7.2.2 Experiment 2: FABRIC Testing

Table 7.4 presents a summary of the overall energy consumption, execution time, and standard deviation values for each implementation and archetype in the distributed FABRIC setup. All p-values reported in the final column ("Energy W (p)") exceed the 0.01 alpha threshold, indicating that the energy consumption data for all implementations is approximately normally distributed (see Chapter 6). However,

for consistency, we applied the same statistical tests as used in the local experiments to allow for equal comparisons of statistical values across both environments.

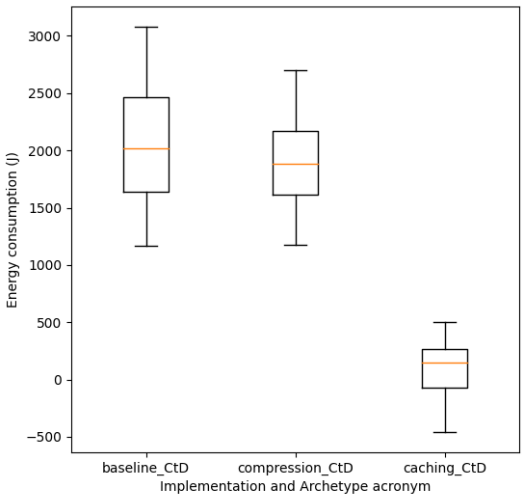
As expected, caching again achieves the lowest energy consumption and fastest execution time in both archetypes. Compared to the local setup (see Table 7.3), caching in FABRIC exhibits similarly strong relative improvements, although the absolute energy savings are even more pronounced due to the higher baseline consumption in the distributed environment. These results indicate the effectiveness and portability of caching as an optimization, especially in multi-node environments like FABRIC.

Compression shows modest improvements in energy consumption and execution time for Compute to Data, although the differences compared to the baseline remain small. These improvements are comparable to those observed in the local setup, where compression also yielded a slight reduction in energy and execution time. In the Data through TTP archetype, however, compression slightly increases energy consumption while only marginally reducing execution time, a pattern that also emerged locally.

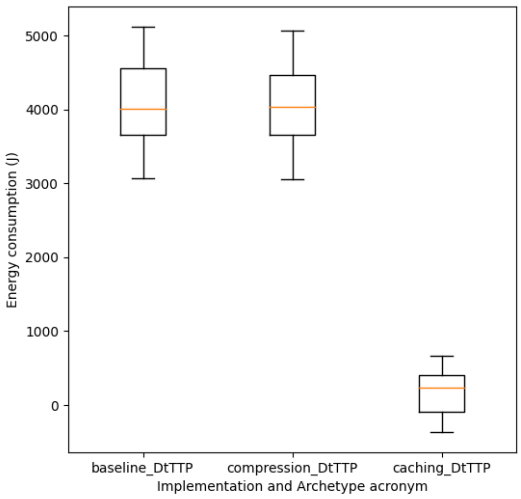
Compute to Data					
Impl.	**Energy (J)**	**SD (J)**	**Time/task (s)**	**SD (s)**	**Energy W (p)**
Baseline	2083.912	543.171	6.537	0.158	0.1782
Compression	1903.897	414.278	6.247	0.094	0.1980
Caching	101.575	245.735	0.347	0.440	0.0904
Data through TTP					
Impl.	**Energy (J)**	**SD (J)**	**Time/task (s)**	**SD (s)**	**Energy W (p)**
Baseline	4071.103	567.294	5.357	0.319	0.1328
Compression	4079.854	592.702	5.258	0.215	0.0718
Caching	179.234	319.115	0.349	0.378	0.0291

Table 7.4: Energy Consumption and Execution Time Results for Implementations and Archetypes Experiment 2

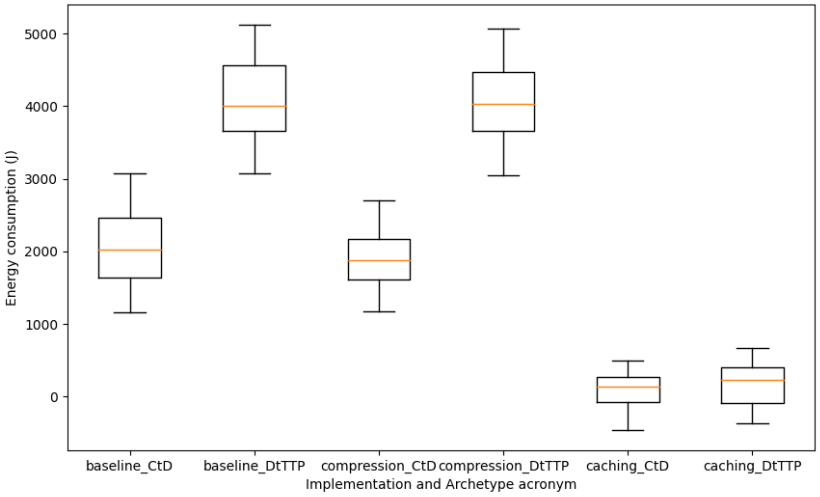
Figure 7.2 shows the energy consumption results as box plots for each archetype and implementation. Overall, the results of Experiment 2 support the same conclusions drawn from the local tests: caching provides substantial energy and performance benefits, while compression exhibits context-dependent outcomes. The following sections will provide a more detailed comparison of each optimization, employing the selected statistical tests to enhance the reliability of conclusions.



(a) Energy Box plot CtD



(b) Energy Box plot DtTTP



(c) Energy Box plot CtD and DtTTP

Figure 7.2: Energy Consumption Box plots for Compute to Data (CtD) and Data through TTP (DtTTP) archetypes Experiment 2

7.3 Compute to Data (CtD)

While Section 7.2 provided an overview and visual representations of the data, this section focuses on analyzing the impact of each optimization compared to the baseline. This analysis not only determines whether a statistically significant difference exists but also evaluates the magnitude of the effect on energy consumption and execution time.

This section specifically examines the Compute to Data archetype. Each section below presents a comparative analysis of an optimization against the baseline, detailing differences in results. For each optimization, the analysis includes a summary table outlining its impact on energy consumption and execution time, followed by a direct comparison of the mean differences between implementations. Finally, statistical tests are performed on the full dataset for each implementation to assess both the significance of the observed differences and the corresponding effect sizes.

7.3.1 Experiment 1: Local Testing

This section presents the comparative results of both caching and compression against the baseline for the Compute to Data archetype in the local (Experiment 1) setup.

7.3.1.1 O1: Caching Requests

Table 7.5 presents the comparison between caching and the baseline for the Compute to Data archetype. The results indicate not only statistical significance but also very high positive improvements in both energy consumption and execution time. In other words, caching results in a substantial reduction in both energy usage and execution duration.

Metric	Mean Difference (%)	MWU (p)	Significant	RBC	Effect
Energy	98.68%	$1.069e-17$	Yes	1.000	Very high, positive
Time	94.77%	$1.069e-17$	Yes	1.000	Very high, positive

Table 7.5: Caching versus Baseline for Compute to Data (CtD) Experiment 1

7.3.1.2 O2: Transferred Data Compression

Table 7.6 presents the comparison between transferred data compression and the baseline for the Compute to Data archetype. The results indicate slight improvements in both energy consumption and execution time. However, these improvements are not statistically significant, suggesting that the impact of compression in this scenario is minimal.

Metric	Mean Difference (%)	MWU (p)	Significant	RBC	Effect
Energy	7.46%	0.423	No	0.095	Slight, positive
Time	3.60%	0.564	No	0.069	Slight, positive

Table 7.6: Compression versus Baseline for Compute to Data (CtD) Experiment 1

7.3.2 Experiment 2: FABRIC Testing

This section presents the comparative results of both caching and compression against the baseline for the Compute to Data archetype in the distributed (Experiment 2) setup.

7.3.2.1 O1: Caching Requests

Table 7.7 presents the comparison between caching and the baseline for the Compute to Data archetype in the FABRIC environment. As in the local setup, caching leads to a statistically significant and very large positive effect on both energy consumption and execution time. These results confirm that caching consistently provides substantial performance and energy efficiency benefits, even in a distributed, multi-node Kubernetes environment. The effect size, as measured by the rank biserial correlation (RBC), reaches its maximum value of 1.000 for both metrics, further highlighting its strong impact.

Metric	Mean Difference (%)	MWU (p)	Significant	RBC	Effect
Energy	95.12%	$5.396e-19$	Yes	1.000	Very high, positive
Time	94.67%	$5.396e-19$	Yes	1.000	Very high, positive

Table 7.7: Caching versus Baseline for Compute to Data (CtD) Experiment 2

7.3.2.2 O2: Transferred Data Compression

Table 7.8 shows the results for the compression implementation compared to the baseline for Compute to Data in FABRIC. Interestingly, compression shows slight improvements in both energy consumption and execution time. However, only the improvement in execution time is statistically significant. The

energy difference, while positive, is small and does not pass the significance threshold. Finally, the effect size for execution time is high (`RBC = 0.897`), indicating a meaningful benefit in performance in this scenario.

Metric	Mean Difference (%)	MWU (p)	Significant	RBC	Effect
Energy	8.63%	0.124	No	0.181	Slight, positive
Time	4.43%	$2.873e-14$	Yes	0.897	High, positive

Table 7.8: Compression versus Baseline for Compute to Data (CtD) Experiment 2

7.4 Data through TTP (DtTTP)

This section compares each optimization to the baseline for the Data through TTP (DtTTP) archetype, similar to the analysis conducted for the Compute to Data archetype in the previous section.

7.4.1 Experiment 1: Local Testing

In this section, we present the comparative results of both caching and compression against the baseline for the Data through TTP archetype in the local (Experiment 1) setup.

7.4.1.1 O1: Caching Requests

Table 7.9 presents the comparison between caching and the baseline for the Data through TTP archetype. The results indicate a very high positive impact on both energy consumption and execution time. Caching significantly reduces energy consumption and execution time, with a very strong statistical significance. The effect size (RBC) for both metrics is 1.000, categorizing the impact as very high positive.

Metric	Mean Difference (%)	MWU (p)	Significant	RBC	Effect
Energy	96.66%	$3.031e-16$	Yes	1.000	Very high, positive
Time	93.00%	$3.031e-16$	Yes	1.000	Very high, positive

Table 7.9: Caching versus Baseline for Data through TTP (DtTTP) Experiment 1

7.4.1.2 O2: Transferred Data Compression

Table 7.10 presents the comparison between transferred data compression and the baseline for the Data through TTP archetype. Unlike caching, compression exhibits a moderate negative impact on energy consumption, which is statistically significant. Specifically, energy consumption increased moderately, while execution time exhibited a slight increase, though the latter was not statistically significant.

Metric	Mean Difference (%)	MWU (p)	Significant	RBC	Effect
Energy	-14.82%	$6.830e-05$	Yes	-0.505	Moderate, negative
Time	-3.18%	0.485	No	-0.089	Slight, negative

Table 7.10: Compression versus Baseline for Data through TTP (DtTTP) Experiment 1

7.4.2 Experiment 2: FABRIC Testing

In this section, we present the comparative results of both caching and compression against the baseline for the Data through TTP archetype in the distributed (Experiment 2) setup.

7.4.2.1 O1: Caching Requests

Table 7.11 presents the comparison between caching and the baseline for the Data through TTP archetype in the FABRIC environment. As with the Compute to Data archetype, caching yields a very high positive

effect on both energy consumption and execution time. The statistical significance is extremely strong ($p = 3.535e{-}17$), and the effect size is maximal (RBC = 1.000) for both metrics. These results confirm that the caching optimization remains effective even in more complex, multi-party communication scenarios such as Data through TTP. Finally, the results demonstrate that caching not only reduces the energy cost of repeated requests but also greatly improves response time, even in distributed settings.

Metric	Mean Difference (%)	MWU (p)	Significant	RBC	Effect
Energy	95.59%	$3.535e-17$	Yes	1.000	Very high, positive
Time	93.47%	$3.535e-17$	Yes	1.000	Very high, positive

Table 7.11: Caching versus Baseline for Data through TTP (DtTTP) Experiment 2

7.4.2.2 O2: Transferred Data Compression

Table 7.12 compares the transferred data compression implementation with the baseline for the Data through TTP archetype in the FABRIC environment. Unlike in the local experiment, where compression increased energy usage significantly, the difference in energy consumption here is negligible (a decrease of only 0.21%) and not statistically significant.

However, execution time is slightly improved by 1.85%, and this result is statistically significant. The effect size is categorized as low positive (RBC = 0.321), suggesting a low benefit in response time while maintaining energy neutrality.

Metric	Mean Difference (%)	MWU (p)	Significant	RBC	Effect
Energy	-0.21%	0.918	No	-0.012	Slight, negative
Time	1.85%	0.004	Yes	0.321	Low, positive

Table 7.12: Compression versus Baseline for Data through TTP (DtTTP) Experiment 2

7.5 Energy & Time Correlation

This section explores the relationship between execution time and energy consumption across all experiments. While earlier sections focused on analyzing optimizations individually, here we examine whether execution time can reliably predict energy efficiency. The following sections provide correlation analyses for both the local and FABRIC environments, supported by scatter plots and statistical metrics.

7.5.1 Experiment 1: Local Testing

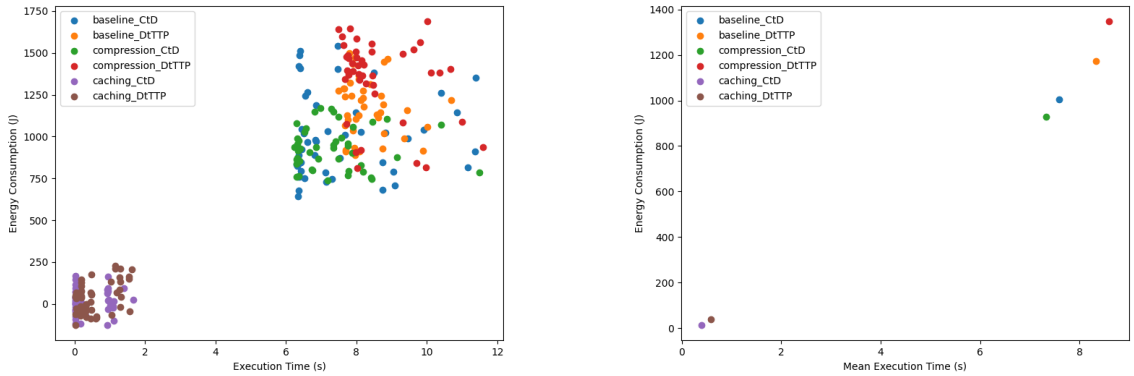
Figure 7.3 illustrates the correlation between energy consumption and execution time across all individual experiments. The results indicate a moderate positive correlation between these two metrics, with a Kendall tau coefficient of $\tau = 0.553$ (see Figure 7.3a). This suggests that, in general, longer execution times tend to be associated with higher energy usage. However, the scatter of the data also shows that this relationship is not perfectly linear, and shorter execution times do not always result in the lowest energy consumption for every individual experiment run.

Interestingly, when analyzing the mean execution times per experiment (see Figure 7.3b), a clearer trend emerges: shorter average execution times more consistently correspond to lower average energy consumption. This supports the general assumption that optimizing execution time can also reduce energy usage, especially when averaged across multiple tasks.

The Kendall tau value is not displayed for the mean-based plot because the dataset contains too few mean values to compute a statistically meaningful correlation—each experiment provides only one mean value for execution time and energy consumption, resulting in an insufficient sample size for the algorithm.

Figure 7.3a also reveals a subtle but important detail in the caching results. The scatter plot shows a small subset of caching experiments with slightly elevated average execution times. This occurs in cases where one or more of the seven runs in an experiment triggered a cache miss—typically during the first request. A cache miss means that no previously stored result is available, and the request must be

executed in full: the required microservices are deployed, the job is carried out, and the result is stored in the cache (see Section 5.1). Subsequent requests within the same experiment then benefit from this cached result, returning almost immediately and consuming far less energy. As a result, experiments with a cache miss exhibit higher average execution times than experiments where all seven requests were served directly from the cache.



(a) All, tau=0.553 (Moderate correlation) **(b) Mean**

Figure 7.3: Energy Consumption and Time Scatter Plots for Compute to Data (CtD) and Data through TTP (DtTTP) Experiment 1

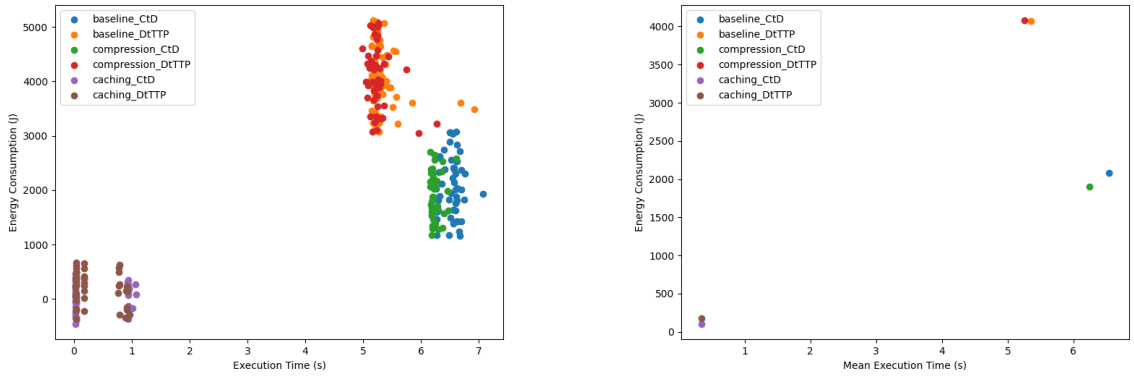
7.5.2 Experiment 2: FABRIC Testing

Figure 7.4 visualizes the relationship between energy consumption and execution time for all implementations in Experiment 2. In contrast to Experiment 1, the correlation between energy and execution time is noticeably lower, with a Kendall tau value of $\tau = 0.222$ (see Figure 7.4a). This indicates a low positive correlation: while there is a general tendency for longer execution times to be associated with higher energy consumption, the relationship is less pronounced and more variable in the FABRIC environment.

As shown in Figure 7.4a, caching implementations cluster at the lower-left part of the plot, reflecting both low execution times and energy consumption. Similar to Experiment 1, a few caching experiments show slightly elevated mean execution times. Other configurations, particularly the baseline and compression implementations, are more dispersed. This demonstrates that faster execution time does not always imply lower energy use for every individual run.

However, compared to the local environment in Experiment 1 (see Figure 7.3a), the results in FABRIC are clearly more stable. The points in each configuration are more tightly grouped, indicating lower variability between individual experiment runs. This reflects the improved stability of the FABRIC infrastructure.

Figure 7.4b plots the average energy consumption and execution time for each experiment. Contrary to what might be expected from Experiment 1, there is no clear downward trend in this mean-based plot. In fact, the orange and red dots show lower average execution times than the blue and green dots, but still consume more energy. This highlights that faster execution time alone is not a reliable predictor of energy efficiency in distributed setups such as FABRIC.



(a) All, tau=0.222 (Low correlation) **(b) Mean**

Figure 7.4: Energy Consumption and Time Scatter Plots for Compute to Data (CtD) and Data through TTP (DtTTP) Experiment 2

7.6 Additional Experiments

In this section, we present a set of additional experiments that were conducted to further investigate unexpected or unexplained observations in earlier results. While the primary experiments were designed to evaluate the impact of specific optimizations in DYNAMOS, these supplementary tests aim to isolate underlying infrastructure-related factors and explore possible causes for discrepancies in system behavior—particularly those observed between the local and distributed environments.

Due to time constraints, these experiments are intentionally limited in scope and depth. Nonetheless, they serve as a valuable diagnostic layer and provide insight into system-level performance, deployment stability, and orchestration behavior under different architectural setups. Importantly, they offer potential explanations for certain results seen in earlier sections—such as the occasionally superior performance of the Data through TTP archetype over Compute to Data in the FABRIC testbed.

The first section investigates infrastructure-level performance differences between the local single-node cluster and the distributed multi-node FABRIC setup using dedicated benchmarking tools. The second section revisits differences in archetype execution time when modifying data provider configurations, building on the methodology used in our preliminary experiments (see Section 3.2.2) and using a reduced number of iterations to balance detail and feasibility. The final section explores the internal flow of a DYNAMOS data request by comparing service traces for Compute to Data and Data through TTP in the FABRIC environment. Together, these additional experiments support a deeper understanding of system behavior and offer valuable context for interpreting the results presented throughout this chapter.

7.6.1 Environment Cluster Comparison

This section explains the approach taken to investigate and compare the performance characteristics of the single-node local Kubernetes environment and the multi-node FABRIC testbed environment. The primary motivation for this comparison stems from earlier experimental results, which revealed noticeable differences in overall execution time between the two setups. In an effort to gain further insight into the root causes of these discrepancies, an additional benchmarking experiment is conducted using infrastructure-level performance testing tools.

To explore the observed performance gap, we designed a dedicated benchmark experiment aimed at isolating infrastructure-related factors that might explain the variation in execution time. Specifically, this experiment evaluated how Kubernetes orchestration and pod lifecycle behaviors differ in local (Docker Desktop) versus distributed (FABRIC) environments. By employing a standardized benchmarking utility across both setups, the goal is to quantify these differences using low-level Kubernetes metrics and assess their impact on performance.

7.6.1.1 Performance Testing Tools

Performance benchmarking in Kubernetes environments can be broadly categorized into two domains: system-level and application-level testing. System-level tools, such as Kube-burner[1], are designed to assess the performance of the underlying infrastructure—including Kubernetes orchestration, CPU and memory utilization, pod scheduling efficiency, and overall cluster responsiveness. In contrast, application-level tools like Locust[2] evaluate software behavior under load, focusing on aspects such as API throughput, user flow simulation, and web service performance [115, 116].

In the context of this study, our objective is to investigate whether infrastructure-level differences between the local and FABRIC environments could explain the variations in execution time observed in earlier experiments. Consequently, we selected Kube-burner as our benchmarking tool, given its explicit focus on Kubernetes infrastructure and its alignment with the goals of this diagnostic experiment.

Kube-burner is an open-source benchmarking utility specifically designed for Kubernetes. It facilitates the orchestration of large-scale workloads by automating the creation, deletion, and patching of resources such as pods and deployments. This allows for the simulation of realistic application-level stress scenarios at the system layer. Its high degree of configurability and support for multiple Kubernetes distributions make it particularly effective for comparative testing across diverse environments, such as local single-node clusters and multi-node testbeds like FABRIC [115].

Kube-burner has also been successfully applied in academic and industry research settings to evaluate the performance characteristics of Kubernetes clusters [116], further validating its suitability for the experiment conducted in this thesis.

7.6.1.2 Performance Testing Setup & Design

Given the extensive nature of the main experiments in this thesis, this additional benchmark is designed to be minimal in scope. Its primary purpose is not to evaluate optimizations, but to explore a potential explanation for the performance differences observed between the local and FABRIC environments. To this end, we conduct a focused experiment using two representative infrastructure-level benchmarking workloads from Kube-burner:

- `kubelet-density`[3]: This lightweight workload is designed to measure baseline orchestration latency in Kubernetes clusters. It deploys a sequence of ephemeral pods based on the minimal `pause:3.1` container image. These pods do not perform application logic but rather serve to isolate and evaluate Kubernetes infrastructure performance, such as pod scheduling latency, container startup time, and resource provisioning overhead. The workload progressively increases the pod creation rate (QPS) across job iterations, thereby testing the responsiveness of the cluster's control plane under growing orchestration pressure.
- `kubelet-density-heavy`[4]: This workload introduces a more realistic application scenario consisting of a client-server architecture. Each iteration deploys a basic application that initiates PostgreSQL queries to a co-scheduled database pod via a Kubernetes service. This setup introduces multiple system components—such as DNS resolution, service routing, and database connectivity—into the measurement scope. It better approximates real-world microservice deployments and helps assess the network, readiness, and service discovery subsystems in both local and distributed Kubernetes environments.

Each workload is executed on both the local Docker Desktop Kubernetes cluster and the distributed, multi-node FABRIC testbed. Performance metrics are collected using Kube-burner's built-in `podLatency` measurement, which captures key timestamps for pod lifecycle stages: `PodScheduled`, `Initialized`, `ContainersReady`, and `Ready` [115, 116]. To ensure comparability, all experiments use identical configuration templates and runtime parameters, thereby isolating infrastructure as the only variable.

Following the methodology of Ramadan *et al.* [116], each workload is executed five times in both environments to reduce the impact of outliers and ensure result reliability.

7.6.1.3 Performance Testing Results

We execute both workloads—`kubelet-density` and `kubelet-density-heavy`—on the local Kubernetes cluster and on the distributed FABRIC testbed. Tables 7.13 and 7.14 present a comparative summary

[1] `https://kube-burner.github.io/kube-burner/v1.20.6/`
[2] `https://locust.io/`
[3] https://github.com/kube-burner/kube-burner/tree/main/examples/workloads/kubelet-density
[4] https://github.com/kube-burner/kube-burner/tree/main/examples/workloads/kubelet-density-heavy

of key pod lifecycle metrics, using the mean values of five repetitions per environment.

Kube-burner collects timestamps associated with specific Kubernetes pod lifecycle events. According to the official documentation[5], these events are defined as follows:

- PodScheduled: The point at which the pod is assigned to a node.
- Initialized: The moment when all init containers within the pod have successfully started.
- ContainersReady: Indicates that all containers in the pod are ready to start serving.
- Ready: Signifies that the pod is ready to receive traffic and be registered in any matching Kubernetes services for load balancing.

In both tables, a value of `0` indicates that the operation completed in less than one millisecond. Kube-burner rounds these measurements, and therefore such ultra-fast events are recorded as zero.

Metric	Local (ms)			FABRIC (ms)		
	99th	Max	Avg	99th	Max	Avg
PodScheduled	0	0	0	0	0	0
Initialized	0	0	0	0	0	0
ContainersReady	1900	2000	1340	1200	1200	880
Ready	1900	2000	1340	1200	1200	880

Table 7.13: Mean Pod Lifecycle Metrics using `kubelet-density` Workload

Metric	Local (ms)			FABRIC (ms)		
	99th	Max	Avg	99th	Max	Avg
PodScheduled	300	600	30	0	0	0
Initialized	300	600	30	0	0	0
ContainersReady	40300	40400	19190	36600	38600	16540
Ready	40300	40400	19190	36600	38600	16540

Table 7.14: Mean Pod Lifecycle Metrics using `kubelet-density-heavy` Workload

The results show that the FABRIC Kubernetes environment consistently outperforms the local setup in terms of orchestration latency. In the `kubelet-density` workload, which isolates infrastructure orchestration performance, both environments perform identically in pod scheduling and initialization phases. However, FABRIC demonstrates a significantly lower average time for `ContainersReady` and `Ready`, suggesting improved efficiency in container startup and readiness signaling.

In the more complex `kubelet-density-heavy` workload, which incorporates service discovery and database interactions, the differences become more pronounced. The FABRIC cluster completes both `ContainersReady` and `Ready` stages over 2.6 seconds faster on average than the local cluster. Furthermore, while the local cluster experiences measurable delays even in pod scheduling and initialization, these stages complete in under one millisecond on FABRIC, as indicated by the rounded-zero values.

7.6.2 Archetype Setup Comparison

In this section, we compare the execution times for different DYNAMOS archetype setups across both the local and FABRIC environments. Two archetypes are evaluated: Compute to Data (CtD) and Data through TTP (DtTTP). Each archetype is tested in both a standard and a modified configuration. In the standard configuration, a single data provider (UVA) is used. In the modified configuration, two data providers (UVA and VU) are included.

Initially, DYNAMOS did not support multiple data providers for the Compute to Data archetype in the old setup (without the API-gateway). Only one pod was created per request, directly through the agent (e.g., UVA or VU). However, for the Data through TTP archetype, it was possible to specify multiple data providers, leading to the creation of multiple pods—e.g., one for SURF, UVA, and VU—if

[5] `https://kube-burner.github.io/kube-burner/v1.20.6/measurements/#metrics`

all providers were listed. With the introduction of the new API-gateway-based setup, Compute to Data could also support multiple data providers. However, due to instability and inconsistency observed with this setup—such as varying response sizes and occasional empty responses—we reverted to the old configuration for more consistent experimental results (see Section 3.2.2).

Notably, even in the modified Compute to Data setup under the old configuration, adding multiple data providers to the request does not change the behavior: only one pod is created per request. This is logical, since in the old setup, requests are made directly to a single agent—such as UVA or VU—rather than coordinated through an intermediate gateway. In contrast, the newer API-gateway-based setup enables inter-agent coordination and pod spawning based on the list of available providers. The Data through TTP archetype, however, does reflect these modifications in the old setup by instantiating additional pods when multiple data providers are specified, since the request is coordinated through a third party (in this case SURF).

Table 7.15 presents the execution times for each run using the reverted old DYNAMOS setup, while Table 7.16 shows the results using the newer API-gateway-based setup. In all cases, results are reported separately for the local and FABRIC environments.

Run	**CtD Stand (s)**		**CtD Mod (s)**		**DtTTP Stand (s)**		**DtTTP Mod (s)**	
	Local	**FABRIC**	**Local**	**FABRIC**	**Local**	**FABRIC**	**Local**	**FABRIC**
1	6.912	6.573	7.185	6.329	8.156	5.178	9.326	7.062
2	6.551	6.560	6.506	6.363	6.579	5.056	9.125	6.995
3	6.816	6.397	7.683	6.267	9.055	5.051	9.232	7.463
4	6.696	6.452	7.894	6.379	8.246	5.203	9.244	7.226
5	6.624	6.491	6.701	6.283	7.754	5.074	11.558	7.180
6	6.596	6.349	7.752	6.285	8.408	5.100	11.386	7.105
7	6.568	6.321	6.445	6.300	6.797	5.208	12.138	7.405
8	6.482	6.325	6.565	6.313	8.336	5.152	9.082	7.435
9	6.468	6.309	6.927	6.279	6.731	5.267	8.964	7.054
10	7.482	6.362	6.724	6.237	8.659	5.234	8.955	6.981
Avg	6.719	6.414	7.038	6.304	7.872	5.152	9.901	7.191

Table 7.15: Old Setup Execution Times per Run for All Archetype Setups and Environments

Run	**CtD Stand (s)**		**CtD Mod (s)**		**DtTTP Stand (s)**		**DtTTP Mod (s)**	
	Local	**FABRIC**	**Local**	**FABRIC**	**Local**	**FABRIC**	**Local**	**FABRIC**
1	7.091	6.557	9.997	6.464	6.700	5.400	10.205	6.603
2	6.943	6.439	10.195	6.883	9.189	6.278	10.375	5.879
3	6.841	6.358	10.382	6.403	8.247	6.312	9.921	7.501
4	6.856	6.367	10.660	6.351	8.337	5.313	8.494	4.872
5	7.212	6.382	10.140	6.371	7.505	5.298	10.037	6.559
6	7.794	6.413	9.540	6.881	9.176	5.298	9.302	4.985
7	6.750	6.366	9.642	6.430	8.654	5.339	8.617	5.858
8	7.533	6.352	10.633	6.974	8.506	6.274	8.530	7.565
9	7.345	6.328	8.752	6.850	8.270	5.305	9.737	4.917
10	6.396	6.415	11.801	6.497	8.131	5.417	11.251	8.565
Avg	7.076	6.398	10.174	6.610	8.272	5.624	9.647	6.330

Table 7.16: API-Gateway Setup Execution Times per Run for All Archetype Setups and Environments

Results reveal that the local environment exhibits significantly more variance in execution times across runs compared to the FABRIC environment, where execution times are relatively stable. This indicates that the local, single-node setup is more sensitive to load variations. Although a full comparison between the old and newest DYNAMOS versions was not feasible due to time constraints, limited testing supports the decision to revert to the older configuration (described in Section 3.2.2), as this configuration produced more stable results. With the newer API-gateway-based setup, we observe inconsistent results such as fluctuating response sizes (ranging from 43 bytes to over 1.3 MB), or responses occasionally being empty. These inconsistencies are especially pronounced when multiple data providers are specified for the Data through TTP archetype. In contrast, the older configuration consistently returns correct and stable results.

While minor inconsistencies remain in the old setup—particularly in Data through TTP with multiple data providers—they are significantly less pronounced than in the API-gateway-based configuration. This reliability validates our decision to use the more stable, default DYNAMOS setup for all key experiments.

The observed instability when using multiple data providers may result from underlying issues in the DYNAMOS data selection logic. However, analyzing and resolving those mechanisms further falls outside the scope of this thesis.

7.6.3 Analyzing Execution Behavior with Jaeger Tracing

As shown in the results sections, the performance experiments conducted in the FABRIC environment reveal an unexpected result: the Data through TTP (DtTTP) archetype occasionally outperforms the Compute to Data (CtD) archetype in FABRIC. Given that the DtTTP setup involves more communication steps and a more complex chain of services, this observation contradicts initial expectations. However, the archetype-level performance comparison in the previous section did not provide a definitive explanation for this discrepancy. To explore potential reasons behind this behavior, we conducted an additional analysis using distributed tracing via Jaeger.

Figures 7.5 and 7.6 show the Jaeger traces for representative runs of each archetype in the FABRIC testbed. These traces capture the flow of an SQL data request through DYNAMOS and illustrate the sequence and duration of service interactions involved in each execution path.

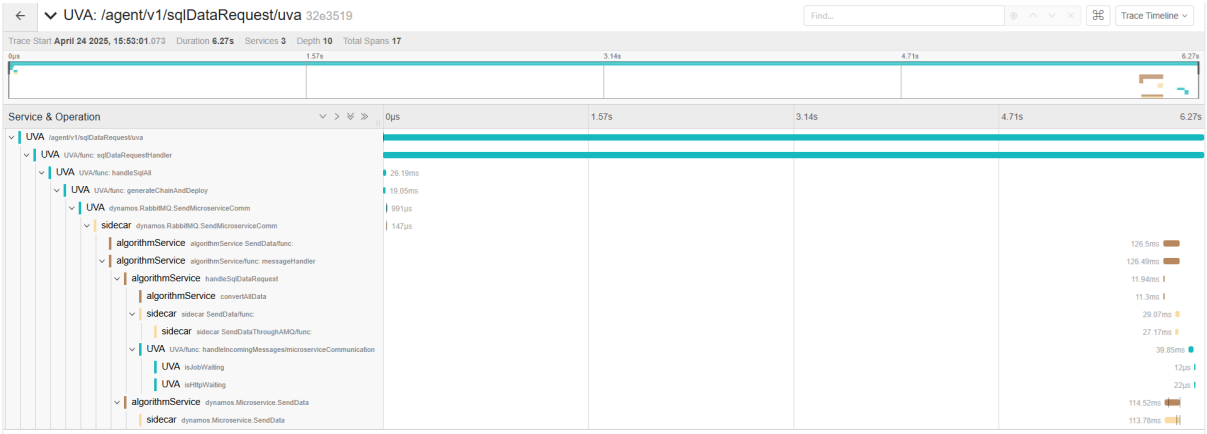


Figure 7.5: Trace of FABRIC SQL Request Compute to Data

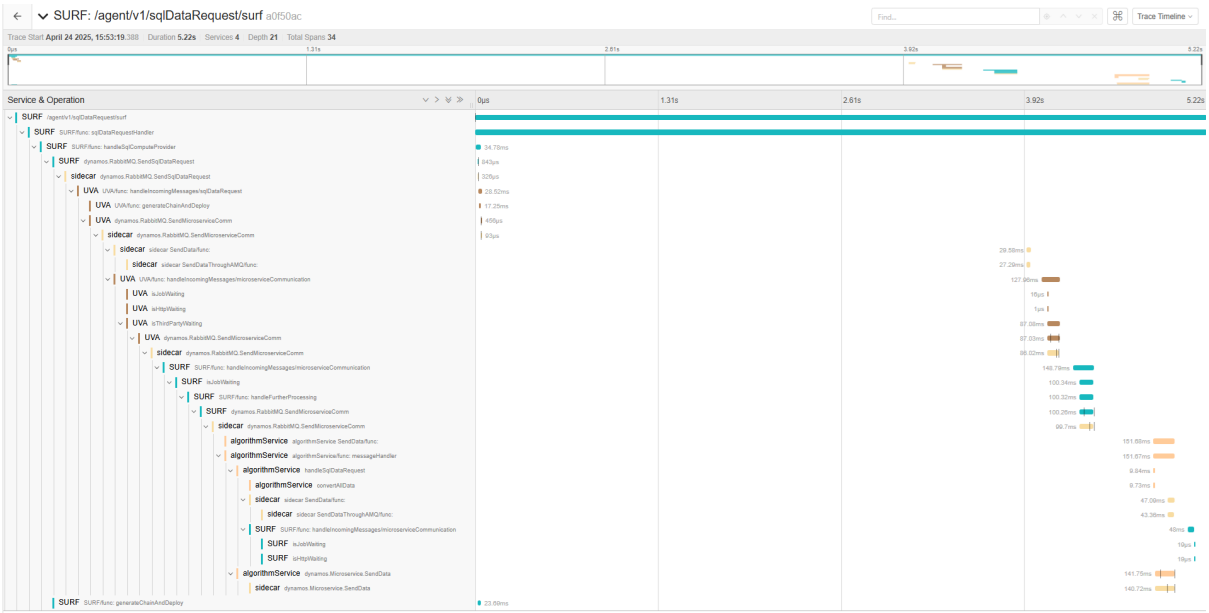


Figure 7.6: Trace of FABRIC SQL Request Data through TTP

In both traces, we observe a noticeable gap after the sidecar `dynamos.RabbitMQ.SendMicroserviceComm` operation. This issue is particularly visible in the trace for Compute to Data (Figure 7.5), where there is a substantial time interval—more than 4 seconds—between the initial microservice deployment and the next observable spans. A similar but shorter delay is present in the Data through TTP trace (Figure 7.6), though in this case, more of the sub-operations appear captured.

This delay precedes the appearance of subsequent spans in the trace and likely corresponds to a waiting period where the message is in transit, or is queued for handling by RabbitMQ. However, it is also possible that this reflects a gap in trace instrumentation—either because the traces from the services are not being correctly emitted, or because Jaeger fails to link them into the full trace graph. It was also mentioned in discussions with the DYNAMOS team that tracing may have become partially broken since a major internal refactor to the new API-gateway setup, which supports this interpretation.

At this point, we do not have a definitive explanation for the origin of this delay, as the available traces do not provide enough information to conclusively determine its cause. However, addressing these issues in more depth would require a full audit and fix of the tracing pipeline within DYNAMOS, which is beyond the scope of this thesis.

Chapter 8

Discussion

In this chapter, we discuss our findings, address the research questions, and provide additional insights gained from this study. Furthermore, we discuss limitations, examine possible threats to the validity of our results, and provide potential future research directions.

8.1 Results Discussion

This section provides a discussion of our findings in Chapter 7, highlighting key insights related to the implemented optimizations, archetype behavior, environment characteristics, and the relationship between execution time and energy consumption.

8.1.1 Implemented Energy Optimizations

In this section, we outline the observed effects of the two implemented energy optimizations—caching and compression—based on results from the experiments.

8.1.1.1 O1: Caching Requests

By storing the results of repeated data requests, caching avoids unnecessary recomputation and service deployment, which are among the main contributors to energy consumption in DYNAMOS. Our experiments consistently demonstrated that this optimization led to a significant reduction in energy consumption and execution time across all archetypes. Furthermore, this effect was clearly observable in both local and FABRIC environments. The fact that caching provides consistent energy savings regardless of environment underlines the value of this optimization and suggests that it is likely to generalize well to other deployments.

Finding 8.1: Caching significantly reduces energy consumption and execution time.

8.1.1.2 O2: Transferred Data Compression

The results for data compression presented mixed outcomes across both experimental environments. In the local environment (Experiment 1), compression generally showed no significant difference in energy consumption and execution time. However, for the Data through TTP archetype, a moderate but statistically significant increase in energy consumption was observed.

Finding 8.2: In a single-node environment, compression does not significantly reduce energy consumption and execution time, and in some cases, it moderately increases energy consumption while generally having no significant impact on execution time.

This result is likely attributed to the additional energy required for compressing and decompressing data between each service. Since every service needs to process the data, it must first be serialized from `google.protobuf.Struct` to `[]bytes`, compressed, then decompressed before further processing. These

steps may offset the energy savings from transmitting smaller payloads, particularly in architectures like Data through TTP where many services are involved.

For the Compute to Data archetype in Experiment 1, a slight reduction in energy consumption was observed; however, this improvement was not statistically significant. In contrast, the Data through TTP archetype exhibited a moderate increase in energy consumption. This discrepancy is likely influenced by system instability issues in DYNAMOS (see Section 8.3.2). One possible explanation is that DYNAMOS occasionally encounters request timeouts due to inefficiencies in inter-service communication. The additional processing overhead introduced by compression and decompression increased latency for the Data through TTP archetype, which may further amplify these instabilities—particularly in the Data through TTP archetype, where more services are involved in data exchange.

In comparison, the Compute to Data archetype appears to be less susceptible to these instabilities, which may explain the absence of a statistically significant negative impact. Furthermore, the average execution time for Compute to Data was lower than its baseline, whereas for Data through TTP, execution time exceeded the baseline. Since we found that execution time in the local environment is correlated with energy consumption, this increased execution time likely contributed to the higher energy consumption observed in the Data through TTP archetype.

In Experiment 2 (FABRIC), the results for compression differed notably. Particularly for the Data through TTP archetype, we observed an increase in performance—both in terms of reduced execution time and a less severe increase in energy consumption compared to the local environment. A plausible explanation is that, in FABRIC, the data is physically transferred between nodes, so reducing its size via compression indeed speeds up transmission. This contrasts with the local environment, where all services reside on the same node and any gains from reduced data size are negated by the compression overhead.

Furthermore, the results in FABRIC appeared more stable, which may also have contributed to these more favorable results. One hypothesis is that requests in FABRIC are distributed across distinct pods on different nodes (e.g., SURF and UVA), allowing for parallel execution and faster results. In contrast, the local setup processes requests on a single node, resulting in higher execution time despite lower energy usage.

Overall, while compression had limited benefit in the local environment, its impact in the FABRIC deployment suggests that compression may be more effective when actual inter-node data transfer is involved, supporting its use in distributed setups.

Finding 8.3: Compression may provide performance benefits in distributed environments with real inter-node data transfers, though energy efficiency gains remain limited.

8.1.1.3 Optimization Comparison

Comparing the two optimizations, caching provided significant improvements in both energy consumption and execution time for all scenarios. In contrast, compression did not yield the same benefits, and in some cases, it even increased energy consumption.

Finding 8.4: Overall, caching proves to be a reliable optimization across all scenarios, whereas the impact of compression remains context-dependent and generally limited.

A key distinction between these optimizations is that caching eliminates the need to repeatedly deploy microservices for identical requests, whereas compression still involves executing the full microservice pipeline for each request. This indicates that repeated microservice deployment is a particularly energy-intensive operation.

Finding 8.5: Deploying the microservice chain is an energy-intensive operation.

This observation is supported by the substantial reduction in energy consumption in the caching implementation, where microservices are only deployed once every 10 minutes rather than for each request. Furthermore, since the baseline and compression implementations generally showed no significant differences in energy consumption, the primary contributing factor to the observed savings in caching appears to be the avoidance of frequent microservice deployments.

In the FABRIC environment, these findings were further reinforced. Despite being deployed in a more complex, distributed setting, caching continued to demonstrate significant energy savings. This suggests that the overhead of repeatedly spinning up microservice chains—especially across multiple nodes—is even more pronounced in distributed systems. On the other hand, the compression optimization showed slightly improved performance in FABRIC due to real inter-node data transfer benefits, but still failed to offset the overhead from compression and decompression logic.

Importantly, compression does not bypass the orchestration and deployment steps inherent to microservice execution. Every request, even when using compression, goes through the full DYNAMOS workflow: policy enforcement, composition generation, job scheduling, and service instantiation. This leads to repeated allocation of computing resources and container scheduling, which are energy-intensive operations, especially evident in FABRIC with multiple worker nodes involved.

Ultimately, caching offers clear advantages by reducing the number of microservice instantiations and thereby minimizing orchestration overhead. In contrast, compression, while potentially beneficial for performance in distributed setups, does not provide comparable energy savings and may even introduce new inefficiencies due to added CPU usage.

Finding 8.6: Caching's energy savings are primarily attributed to the avoidance of repeated microservice deployments, a cost that remains substantial even in distributed environments.

8.1.2 Archetype Comparison: Compute to Data & Data through TTP

The comparison between the Compute to Data (CtD) and Data through TTP (DtTTP) archetypes revealed consistent differences in energy consumption, particularly in the local environment. As shown in Table 7.3, the DtTTP archetype consistently exhibited higher energy consumption and execution time than CtD across all implementations. This result aligns with initial expectations: DtTTP introduces an additional third party (e.g., SURF) between data providers and consumers, increasing the number of deployed services and communication steps. Each request involves more pods and data transfers, which leads to higher resource usage and consequently more energy consumption.

Finding 8.7: The Data through TTP archetype consumes more energy, likely due to a higher number of data transfers and additional service deployments compared to the Compute to Data archetype.

In Experiment 2 (FABRIC), however, the results were more nuanced. While DtTTP continued to consume more energy overall, it unexpectedly outperformed CtD in terms of execution time across multiple runs and configurations. This behavior contradicts initial assumptions, since DtTTP involves a more complex communication chain and additional microservice deployments.

To further investigate this behavior, we conducted an additional set of experiments comparing execution times across different archetype configurations and setups (see Sections 7.6.2). These experiments revealed that in the local environment, adding more pods—such as when involving multiple data providers—led to increased execution time due to shared resource contention on a single node. In contrast, the FABRIC environment maintained consistent and low execution times, even with additional pods, suggesting that its multi-node architecture enables better resource distribution and mitigates contention.

These findings support the hypothesis that DtTTP may benefit from distributed execution in FABRIC, allowing services like SURF and UVA to run in parallel. Meanwhile, CtD requests are often processed sequentially on a single node, limiting their responsiveness. However, although several hypotheses were explored—including parallelism, pod scheduling, and network behavior—a definitive explanation for DtTTP's superior execution time in FABRIC could not be determined within the scope and timeline of this thesis.

To complement the performance data, distributed tracing via Jaeger was employed to analyze the execution behavior of both archetypes (see Section 7.6.3). These traces revealed unexplained delays, especially in the CtD setup, following the microservice communication step. This gap likely corresponds to queuing or orchestration latency, though missing spans and incomplete instrumentation limit our ability to draw firm conclusions. Nonetheless, the consistent presence of this delay in CtD traces strengthens the hypothesis that infrastructure-level behavior—such as message delivery latency or pod scheduling inefficiencies—may significantly influence system performance.

Finding 8.8: In distributed environments, infrastructure-level factors such as scheduling, resource allocation, and communication overhead could outweigh architectural complexity in determining system performance.

Overall, these findings highlight that while DtTTP's higher energy usage is consistent with its greater architectural complexity, its occasional performance advantage in distributed environments like FABRIC underscores the need to consider both system design and deployment context. Performance evaluations in microservice-based systems must account not only for the intended architecture, but also for the orchestration, scheduling, and runtime dynamics of the underlying infrastructure.

8.1.3 Environment Comparison: Local & FABRIC

The experiments in this thesis were conducted in two distinct environments: a local single-node Kubernetes cluster (Docker Desktop on WSL2) and a distributed, multi-node FABRIC testbed deployment. This section provides a comparison of these environments, highlighting key differences in energy consumption, performance, and system stability.

8.1.3.1 Differences in Energy Consumption

A consistent pattern observed across all experiments was that the FABRIC environment consumed significantly more energy than the local setup. This discrepancy is primarily attributed to the differences in cluster topology. The local environment consisted of a single-node Kubernetes cluster running on a virtual machine (WSL2), whereas the FABRIC setup spanned multiple nodes, each hosting one or more DYNAMOS components such as agents or SQL services (see Section 5.3.4).

In a multi-node environment, distributed orchestration and service deployment introduce overheads not present in a single-node setup. For example, services that communicate across nodes incur network transmission costs and synchronization delays, potentially contributing to increased energy consumption. Furthermore, the larger number of nodes deployed in the FABRIC Kubernetes environment results in greater resource usage overall, especially for architectures like Data through TTP that require instantiating multiple pods (e.g., SURF and UVA) for each data exchange request.

Finding 8.9: The experimental environment—whether local or distributed—has a significant influence on energy consumption measurements. Distributed setups like FABRIC introduce additional overhead through inter-node communication, increased pod orchestration, and service distribution, all of which contribute to higher energy usage compared to single-node local environments.

8.1.3.2 Infrastructure Performance Differences

To explore infrastructure-level performance disparities, additional benchmarking experiments were conducted using Kube-burner (see Section 7.6.1). These tests revealed that the FABRIC environment consistently outperformed the local setup in terms of orchestration latency. For both the lightweight `kubelet-density` and the more complex `kubelet-density-heavy` workloads, pod lifecycle metrics such as `ContainersReady` and `Ready` were significantly faster in FABRIC.

These results indicate that the FABRIC cluster exhibits better resource distribution, more efficient pod scheduling, and reduced contention. This infrastructure efficiency likely contributes to the observed improvements in execution time for some archetypes, despite higher energy usage.

8.1.3.3 System Stability Differences

Beyond orchestration efficiency, system stability was a key differentiator between the environments. In the local setup, DYNAMOS exhibited frequent instability—especially within the Data through TTP archetype—resulting in greater execution time variability and a higher number of invalid results due to request failures. This is evident in the elevated "Non-200 Status" counts in Table 7.1. These issues are likely caused by resource contention: all services shared a single compute node, leading to fluctuating CPU and memory availability under concurrent workloads.

In contrast, the FABRIC environment delivered significantly more stable performance. Execution times were more consistent across repeated runs for both archetypes (see Section 7.6.2), and the number of

failed requests was markedly lower (see Table 7.2). Although the number of repetitions varied slightly due to preventive over-execution, actual crashes or non-200 status responses were rare. This was particularly true for the caching implementation, which produced no invalid repetitions across archetypes.

Results in FABRIC were also more tightly clustered, indicating lower variability compared to the local setup (see Figures 7.3a and 7.4a). This suggests a more predictable infrastructure, where distributed workloads avoid the contention seen in single-node environments. Moreover, while local results sometimes deviated from normality, FABRIC data consistently met statistical assumptions, reinforcing its reliability as a testbed.

The evidence thus clearly supports the conclusion that FABRIC provided a more stable and predictable foundation for experimentation. This enhanced system behavior improved both system reliability and the integrity of energy and performance metrics in Experiment 2. Although these hypotheses were not further validated due to time constraints, the observed trends strongly indicate that distributed orchestration and resource isolation play a key role in achieving system stability.

Finding 8.10: The FABRIC deployment exhibited significantly greater stability than the local environment, likely due to improved orchestration, resource separation, and parallelism. Tighter result clustering and fewer failed runs further confirm FABRIC's reliability.

8.1.4 Execution Time & Energy Consumption Correlation

To clarify the relationship between energy and time, we analyzed the correlation between execution time and energy consumption across both the local and distributed FABRIC environments.

Previous research [18, 117, 118] suggests that energy consumption is influenced by execution time. However, literature presents varying conclusions regarding the relationship between performance and energy efficiency. Some studies, such as [44], argue that increased performance leads to reduced energy consumption. Others, including [18, 60], suggest that faster execution does not necessarily equate to lower energy consumption. Additional research [119–121] further supports the notion that execution speed is not always a reliable indicator of energy efficiency. Factors such as memory usage, CPU load, and thread management can all influence the energy impact of optimizations.

In the local, single-node environment, we observed a moderate correlation between execution time and energy consumption. This suggests that, in scenarios where all microservices share the same physical machine, reducing runtime often leads to lower energy use. This could be attributed to the fact that a single compute context reduces variation in how resources are allocated and shared, making runtime a more reliable indicator of underlying energy expenditure.

However, this relationship does not hold as strongly in the distributed FABRIC environment. There, we found only a low correlation between execution time and energy consumption. While certain configurations, such as caching, consistently performed well across both metrics, other implementations (e.g., compression for the Data through TTP archetype) achieved shorter execution times but consumed significantly more energy. These results demonstrate that energy efficiency in distributed systems is influenced by factors beyond just execution time—such as the number of active nodes, pod scheduling decisions, inter-node communication, and orchestration overhead.

This discrepancy between environments underscores an important implication for practitioners: in distributed Kubernetes-based systems, reducing execution time does not necessarily lead to energy savings. In fact, performance gains may even come at the cost of increased energy usage due to background system activity and coordination overheads. Therefore, optimization strategies that target energy efficiency must take into account not only service-level performance but also deployment topology and infrastructure behavior.

Finding 8.11: Execution time is a moderately reliable proxy for energy consumption in single-node environments, but this relationship weakens significantly in distributed systems.

These insights suggest that energy-aware software design, especially for cloud-native or microservice-based systems, must rely on direct energy measurements rather than simplistic assumptions about performance. While execution time remains an important factor, it is only one dimension in understanding and optimizing energy behavior in modern computing environments.

8.2 Research Questions

The previous sections presented and discussed the results of our experiments. In this section, we answer the research questions established at the beginning of this thesis.

8.2.1 Research Question 1

RQ1 What energy efficiency optimizations are suitable for data exchange archetypes?

This thesis identified energy efficiency optimizations by analyzing the root causes of energy consumption in a data exchange system like DYNAMOS. Using this approach, we systematically selected energy optimizations in Section 4.1, where a set of requirements was defined to limit the scope of optimizations considered. This process led to the identification of nine energy optimizations suitable for data exchange archetypes in general (see Section 4.2). To answer this question, the following energy optimizations were found to be applicable to data exchange archetypes:

- **Batch Processing:** see Section 4.2.1.
- **Caching:** see Section 4.2.2.
- **Data Compression:** see Section 4.2.3.
- **Efficient Communication Protocols:** see Section 4.2.4.
- **Energy Efficient Programming Language:** see Section 4.2.5.
- **Energy Code Smell Refactoring:** see Section 4.2.6.
- **Minimizing Number of Data Transfers:** see Section 4.2.7.
- **Persistent Jobs instead of Ephemeral Jobs:** see Section 4.2.8.
- **Query Execution Optimization:** see Section 4.2.9.

8.2.2 Research Question 2

RQ2 To what extent do these optimizations enhance the energy efficiency of data exchange archetypes?

After identifying suitable energy efficiency optimizations, a second set of requirements was applied to narrow the selection to three optimizations that could be implemented in DYNAMOS (see Section 4.3):

- **O1: Caching Requests:** see Section 4.3.2.
- **O2: Transferred Data Compression:** see Section 4.3.3.
- **O3: Query Execution Optimization:** see Section 4.3.4.

In this thesis, two of these optimizations were implemented in DYNAMOS:

- **O1: Caching Requests:** see Section 5.1.
- **O2: Transferred Data Compression:** see Section 5.2.

To answer RQ2, these optimizations were validated through experiments to assess their impact on energy consumption in DYNAMOS across multiple environments (see Chapter 7). The findings are as follows:

- **O1: Caching Requests:** Caching consistently showed a very high positive impact on both energy consumption and execution time across all experiments and archetypes. By eliminating the need to repeatedly deploy microservices for identical requests, caching reduced system overhead, improved stability, and significantly lowered resource usage. These benefits were clearly observed in both local and FABRIC environments.
- **O2: Transferred Data Compression:** The impact of compression is more nuanced. In most cases, compression showed no statistically significant improvement in energy consumption or execution time. In the local environment, compression even resulted in a moderate negative impact on energy usage for the Data through TTP archetype—likely due to the overhead of repeated serialization, compression, and decompression operations. However, in the FABRIC environment, compression led to a slight improvement in execution time (particularly for Compute to Data), suggesting it may offer performance benefits in distributed setups involving actual inter-node data transfers, albeit without strong gains in energy efficiency.

8.3 Insight & Observations Beyond Research Questions

Throughout the research and development process of this thesis, numerous discussions and reflections have led to additional insights and observations. These insights provide valuable perspectives on the

progress, challenges, and outcomes of this thesis. The following section presents a discussion of these key observations.

8.3.1 DYNAMOS Trust System & Policy

DYNAMOS operates as a policy-driven data exchange system, ensuring that data-sharing decisions align with predefined rules. It also functions as a trust system, where scenarios such as sharing data through a Trusted Third Party (TTP) can be configured. Throughout the implementation process, we considered potential implications of our optimizations on the trust and policy mechanisms in DYNAMOS, leading to several important insights.

The compression implementation introduces minimal risks, as it primarily involves compressing and decompressing transferred data without altering the underlying trust system. However, caching presents more significant considerations. Since caching stores request results, which may include sensitive data, certain risks emerge that must be mitigated to ensure compliance with DYNAMOS' trust framework. Changing the policy was not in the scope of this implementation, as there were many other things we wanted to do for this thesis. However, potential concerns and solutions include:

- **Data Sensitivity:** Cached data might contain sensitive information, posing privacy and security risks. This could be mitigated by integrating caching rules within the policy framework, allowing policies to determine whether a request result is eligible for caching.
- **Data Consistency:** Cached data may become outdated if the underlying data changes over time. In DYNAMOS, dataset modifications typically require system redeployment, which resets the cache, reducing this risk. Furthermore, implementing a time-to-live (TTL) mechanism, as done in our caching approach, helps maintain cache freshness. Future work could explore dynamic policy-driven cache invalidation strategies for example.
- **Multi-Tenancy Considerations:** If DYNAMOS is extended to support multiple tenants, a separate cache for each tenant may be required to prevent data leakage across different users.

To minimize these risks, caching in DYNAMOS was implemented only after all policy checks had been completed for example. Future research could further refine the integration of caching within the policy framework to enhance security and trust.

8.3.2 Instability of DYNAMOS

Throughout the experiments, DYNAMOS occasionally exhibited instability due to some underlying bugs in the system in the form of request failures, inconsistent behavior, and system crashes. These issues were most often observed in the baseline and compression implementations, where microservices had to be deployed and executed for every request. The instability manifested primarily as failed data requests.

This instability was more prevalent in the Data through TTP archetype, which involves additional service orchestration and inter-node communication. As multiple services must be started and coordinated to fulfill a single request, the probability of encountering a failure—due to timing issues, message queue delays, or deployment contention—increased. These unstable or failed runs were systematically filtered out during anomaly detection, as described in Section 7.1, ensuring that the final results reflected only successful experiment executions.

A key observation from our experiments was that caching significantly improved the reliability of DYNAMOS in all environments. Since caching prevents the need to redeploy microservices for repeated requests—by reusing results for identical calls within a 10-minute TTL—the system experienced fewer component startups, fewer service coordination events, and thus fewer opportunities for failures. As a result, almost no failed requests occurred when caching was enabled (see Tables 7.1 and 7.2), demonstrating its secondary benefit as a stabilizing mechanism beyond its energy and performance advantages.

Interestingly, we also found that DYNAMOS was substantially more stable in the FABRIC environment compared to the local Docker Desktop setup, as discussed in Section 8.1.3.

8.3.3 Selection Process for Energy Efficiency Optimizations

The process of selecting energy efficiency optimizations was a key discussion point in this thesis, requiring careful consideration to ensure a well-defined scope. At the beginning, we found that there is a broad range of energy efficiency optimizations available in the literature, each varying in scope and focus (see Section 2.4). Some optimizations target high-level architectural changes, such as selecting an

energy-efficient deployment platform, while others focus on specific areas, such as language-specific code optimizations.

Given the time constraints and the amount of work we visualized within this thesis, defining a clear scope was challenging. To address this, we adopted a structured approach. First, we identified the root causes of energy consumption in DYNAMOS, which helped us focus on optimizations relevant to data exchange systems. Furthermore, we applied a set of requirements to further narrow the scope to optimizations specifically applicable to data exchange archetypes in DYNAMOS (see Section 4.1).

Another challenge in this selection process was the limited availability of prior research on energy efficiency optimizations in software systems. As noted by Balanza-Martinez *et al.* [6], energy efficiency research remains relatively underdeveloped compared to other fields, as it is still a relatively recent topic. This lack of established methodologies reinforced the importance of using root cause analysis to systematically explore possible optimizations beyond those already documented in the literature.

Furthermore, selecting among different optimizations within our refined scope required additional prioritization. As outlined in Section 4.1, we prioritized optimizations based on their applicability to data exchange archetypes and their potential impact on energy efficiency. To maintain a higher-level focus, we grouped optimizations where appropriate. For example, instead of evaluating individual energy code smells or green code patterns separately, we considered refactoring energy code smells as a broader category. This approach allowed us to effectively manage scope while ensuring a meaningful contribution to energy-efficient data exchange systems.

8.3.4 Optimization Trade-offs

While this thesis primarily measured energy consumption and execution time, optimizations may introduce trade-offs that were not fully explored. For caching, these trade-offs include increased storage requirements due to cached data, additional system complexity and overhead for managing the cache, and the risk of serving outdated data if consistency is not maintained. There are also security concerns, particularly if sensitive data is stored in the cache, which may require encryption or access control mechanisms. Moreover, caching becomes less effective when request patterns involve highly variable or unique queries, leading to lower cache hit rates.

Compression introduces a different set of considerations. It adds computational overhead, as each service must compress outgoing data and decompress incoming data, increasing CPU and memory usage. This process can also lead to additional latency, particularly if the time spent compressing and decompressing outweighs the benefits of reduced data transmission. Furthermore, compression tends to be most effective for large datasets; smaller payloads may not see significant gains and can even result in negligible or negative effects.

While this discussion focuses on the optimizations implemented in this thesis, other potential optimizations identified for data exchange archetypes may introduce additional trade-offs. However, as no precise implementation was explored for those alternatives, their trade-offs remain speculative.

8.3.5 Differences Between Preliminary and Main Experiments

Several key differences exist between the preliminary experiments and the main experimental setup, which may explain variations in results:

- **Involvement of Policy Mechanisms:** In the preliminary experiments, the request approval step (`/requestApproval`) was performed once and reused for subsequent data requests. In contrast, the main experiments incorporated request approval before each data request to reflect the policy-based decision-making inherent to DYNAMOS. While this had no internal effect on execution, it introduced a semantic distinction, as request approval is an integral part of the chosen archetype (see Section 6.4.2).
- **System Load Differences:** The main experiments involved a significantly higher and more constant system load over time compared to the preliminary experiments. This increased load may have influenced energy measurements by introducing more variability due to background processes or other influences, such as computer temperature.
- **Expanded Scope of Container Monitoring:** The preliminary experiments focused on measuring energy consumption for a single container, whereas the main experiments included multiple containers. This broader scope likely impacted the total reported energy consumption, as additional containers were included in the measurements.

8.3.5.1 Energy Consumption of Policy Enforcement

Another noteworthy aspect is the energy consumed of the policy enforcement mechanism in DYNAMOS. Since request approval involves additional computations, it contributes to overall energy consumption. Comparing the preliminary experiments (where request approval was performed once) with the main experiments (where request approval was included for every data request), we observed higher energy consumption in the latter.

However, we could not precisely quantify the additional energy cost of policy enforcement, as the experimental setups differed in multiple aspects. Conducting a controlled comparison—where only the policy enforcement step varies—would be necessary to determine the exact energy overhead. Due to time constraints, such an analysis was not feasible in this thesis, but it remains a valuable avenue for future research.

8.3.6 Costs of Energy Monitoring

The cost of energy monitoring itself must be considered when evaluating the energy efficiency of software systems. While the primary objective of this thesis was to investigate and optimize the energy consumption of data exchange archetypes within DYNAMOS, the energy consumed by the monitoring setup itself was not within the scope of this study. Nonetheless, it constitutes a non-negligible factor in the overall energy profile of such systems.

As detailed in Section 3.2, this thesis focused solely on the energy consumption of containers directly involved in the execution of the data exchange processes. Monitoring-related containers such as `prometheus`, `grafana`, `kepler`, and other general Kubernetes infrastructure components like `coredns` were explicitly excluded from the measurement queries through a strategic container selection process. This ensured the collected data reflected only the impact of the DYNAMOS archetype implementations and not the associated monitoring infrastructure.

However, it is important to recognize that tools like Prometheus, Grafana, and especially Kepler (which performs continuous metric collection and energy estimation using eBPF probes and machine learning models), incur energy overhead themselves. These services were continuously running during both the local and FABRIC-based experiments, and as such, their energy footprint—while excluded from the experiment results—was still part of the real-world setup. In resource-constrained environments or large-scale deployments, the cumulative cost of such monitoring infrastructure may significantly impact overall energy consumption.

Future research could investigate the energy overhead of these monitoring tools in isolation, comparing scenarios with and without energy monitoring to quantify the trade-off. For instance, deploying a minimal system without Kepler or Prometheus, and comparing total node-level energy consumption against a fully monitored setup, would reveal whether the monitoring infrastructure introduces substantial additional load. Moreover, the configuration and deployment pattern of monitoring tools can influence their energy cost. In this thesis, certain monitoring components (e.g., `kepler-exporter`, `node-exporter`) were deployed as `DaemonSets`, running on every node in the cluster. This ensured comprehensive data collection but may have increased energy usage compared to a centralized or selectively enabled monitoring configuration.

8.3.7 Transferability of Findings and Implementations

While this thesis focused on the DYNAMOS platform, several aspects of the implementation and experimental methodology are transferrable to other systems—especially those operating within Kubernetes-based environments. This section reflects on which components of the system and experimentation pipeline can be reused in other contexts and highlights which parts are tightly coupled to DYNAMOS. Although these components were not tested on other systems, the analysis below is based on theoretical compatibility and implementation logic.

8.3.7.1 Energy Monitoring Setup and Reporting Pipeline

The container-level energy measurement and reporting setup adopted for this thesis (see Section 3.1) is broadly applicable to any Kubernetes-based system. Kepler, the energy exporter used in this study, integrates seamlessly into Kubernetes through Helm charts and DaemonSets. The setup pipeline developed as part of this thesis (see Section 3.2 and the GitHub repository for this thesis presented in Chapter 5) includes automated scripts for deploying Kepler and configuring Prometheus to collect power metrics.

This infrastructure is generic and requires only minor adjustments (e.g., node selectors, namespace configurations, containers) to function in other Kubernetes clusters. As such, the energy monitoring pipeline is highly transferable to other systems and can support a wide range of containerized applications.

8.3.7.2 Caching Implementation

The caching mechanism implemented in DYNAMOS was built around Redis, a widely used in-memory data store that supports key-value operations and TTLs (time-to-live). While the specific caching logic—such as key generation and cache lookup—was implemented in DYNAMOS' Go-based services, the Redis deployment and integration into the Kubernetes cluster are not DYNAMOS-specific. This makes the caching infrastructure itself applicable to other systems. Only the in-service cache handling logic (e.g., DYNAMOS-specific cache keys) would need to be re-implemented to align with the application domain of the target system.

8.3.7.3 Compression Implementation

The compression implementation is specific to DYNAMOS, both in terms of programming language (Go) and in how the compression was integrated into the service communication chain. The code performs serialization using the `google.protobuf.Struct` type and compresses the resulting byte stream using the `compress/gzip` package in Go. As such, this implementation is not directly transferrable to systems written in other programming languages or that use different serialization schemes. However, the concept of compressing service payloads remains broadly applicable and can be implemented in other systems using language-specific libraries.

8.3.7.4 Kubernetes Deployment on FABRIC

The Kubernetes cluster used in this thesis was deployed on the FABRIC testbed using Kubeadm for this thesis, a tool for automating multi-node cluster provisioning. This deployment method is broadly applicable and can be reused for setting up Kubernetes clusters in other FABRIC-based projects or in similar testbed environments. However, components such as the Jupyter notebooks used for node setup, custom SSH tunneling configurations, and the associated orchestration of the DYNAMOS workloads are specific to this experiment and testbed. While the principles are transferable, practical use in other systems would require adaptation of the automation scripts and possibly changes in cluster topology or node roles.

8.3.7.5 Final Considerations

While many of the infrastructure and orchestration setups are transferrable, it is important to acknowledge that each system has its own set of requirements, communication protocols, and service dependencies. As such, adaptations would be necessary when applying these setups to different environments. Furthermore, the implementations discussed here have not been tested outside of DYNAMOS. The above transferability conclusions are therefore theoretical and based on logical system architecture, not empirical testing.

8.3.8 Energy Consumption Setup Comparison

A notable difference between the two test environments—local and FABRIC—was the granularity of the energy consumption data collected by Kepler. On the local Docker Desktop setup (running on WSL2), nearly all energy consumption during both idle and active periods was attributed to a single pseudo-container: `system_processes`. For instance, during an active run, `system_processes` reported over 22,000 joules, while all other containers reported zero energy usage—including the actual DYNAMOS services such as `uva`, `surf`, `orchestrator`, and `sql-query`. This pattern was consistent across multiple local runs.

In contrast, the FABRIC deployment produced much more detailed and accurate container-level energy data. Kepler attributed energy consumption not only to the core DYNAMOS data request services (e.g., `sql-query`, `uva`, `surf`) but also to supporting services like `policy-enforcer`, `orchestrator`, and `sidecar`. Furthermore, FABRIC introduced a separate container label: `kernel_processes`, which captured energy used by system-level operations such as interrupt handling, I/O processing, and memory management as explained in Section 5.3.4. These values were consistently present and non-zero—for example, over 9,800 joules were attributed to `kernel_processes` during an active run.

This difference can be explained by the architectural limitations of the WSL-based Docker Desktop environment used locally. In WSL2, the Linux kernel runs in a virtualized context on top of the Windows host OS, limiting Kepler's access to kernel-level metrics. As a result, energy that would normally be distributed across services is aggregated under `system_processes`, leading to inaccurate or overly coarse-grained reporting. Occasionally, Kepler was able to report energy for specific containers on WSL, but this occurred inconsistently and was therefore excluded from analysis.

To account for this discrepancy, all energy measurements on FABRIC explicitly included both `system_processes` and `kernel_processes` to ensure total consumption was accurately reflected. While this introduces a slight asymmetry with the local setup—where `kernel_processes` is not available—it provides a more faithful representation of actual energy usage in a realistic distributed environment, as explained in Section 5.3.4.

8.4 Limitations

This section outlines the key limitations of this thesis. While some of these limitations may have been briefly mentioned earlier, this section provides a comprehensive overview to clearly define the constraints and challenges encountered during this study.

8.4.1 Limited Implemented Optimization Scope

Although a broader set of nine energy efficiency optimizations was identified for DYNAMOS, only two—request caching and transferred data compression—were selected for implementation and evaluation. This limitation was primarily driven by the time constraints associated with the scope and duration of this thesis and the prioritization of deploying DYNAMOS in a distributed system (FABRIC). As discussed in Section 4.1, the selection process considered both the relevance of each optimization and the feasibility of implementing it within the available timeframe.

Other identified optimizations, such as batch processing, may also offer meaningful energy savings. However, their integration would have required more substantial architectural changes or extended development efforts that exceeded the time available for this research. Consequently, the conclusions drawn in this thesis should be interpreted within the context of the two implemented optimizations. While this limits the breadth of empirical validation, the structured approach to identifying root causes and selecting optimizations lays a foundation for future studies to expand on this work.

8.5 Threats to Validity

This section discusses potential threats that could affect the validity of our findings and the measures taken to mitigate these risks.

8.5.1 Limited Energy Optimizations in Literature

A threat to the validity of this thesis lies in the limited body of literature on energy efficiency in software systems. As highlighted by Balanza-Martinez *et al.* [6], this research domain is still underdeveloped. Consequently, the selection of energy optimizations in this study was constrained by the scope and availability of existing research.

This limitation introduces a potential validity threat: if more comprehensive or newer literature becomes available in the future, it may reveal additional optimizations that were not considered in this thesis. Such developments could influence the prioritization or selection of optimizations and potentially lead to different implementation choices or performance outcomes.

To mitigate this threat, we applied a structured methodology for identifying root causes of energy consumption in DYNAMOS. This approach ensured that optimization selection was not based solely on literature availability, but also on a systematic analysis of system behavior. Nevertheless, the absence of a broader literature base may have affected the generalizability of the chosen techniques and their relative effectiveness.

8.5.2 Reliability of Existing Measurement Tools

One of the primary concerns is the reliability of the measurement tools used for energy consumption reporting, as highlighted by Gudepu *et al.* [33]. While Kepler has demonstrated reliable performance in

prior studies (see Section 3.1.2), distributed environments such as Kubernetes introduce challenges. For example, the presence of numerous background processes unrelated to the measured tasks may influence energy consumption data. Furthermore, Kepler employs a software-based approach, estimating energy metrics through models rather than direct measurements, which could introduce a margin of error.

To mitigate these limitations, preliminary experiments were conducted to improve our understanding of Kepler's reporting behavior and to validate the energy measurement methodology before executing the main experiments (see Section 3.2). Furthermore, background processes were minimized where possible (see Chapter 6 and Section 3.2), and the experiments were conducted in two different environments to improve the reliability of our results.

Furthermore, as explained in Section 3.1.3, all experiments were conducted on virtual machines (VMs), which typically do not expose real-time power metrics. As a result, Kepler relies on software estimations rather than hardware-based measurements in these environments. This limitation may impact the precision of the reported energy consumption, particularly in the local Docker Desktop setup.

8.5.3 Instability of DYNAMOS

DYNAMOS exhibited occasional instability due to some underlying bugs (see Section 8.3.2), some of which could not be fully resolved within the scope of this thesis. While efforts were made during this thesis to improve stability in collaboration with other developers, time constraints limited the extent of these improvements.

While this instability may have influenced some of the energy measurements, the impact was mitigated by conducting a high number of experiment repetitions and removing invalid results through anomaly detection (see Section 7.1). Therefore, we believe the results remain reliable despite occasional instabilities in DYNAMOS. Especially in the FABRIC environment, where we observed that DYNAMOS was significantly more stable, these instabilities are unlikely to have had a major impact.

8.5.4 Lack of Normality

The Shapiro-Wilk normality test revealed that not all collected data followed a normal distribution. To address this, we applied non-parametric statistical tests, such as the Mann-Whitney U test, which do not assume normality. These rank-based tests are known to be robust even when the normality assumption is violated.

The lack of normality in the data may have been influenced by external factors beyond our control. Background processes in Kubernetes, despite mitigation efforts, could have introduced variability in energy consumption. Furthermore, the instability of DYNAMOS (see Section 8.3.2) may have contributed to irregularities in execution time and energy data.

It is important to note that the normality issue primarily affected the local environment. The results from the FABRIC experiments were not subject to this limitation, as the data in those tests passed the Shapiro-Wilk test consistently.

One possible way to counteract this issue would be to increase the number of experiment repetitions. However, given that our experiment sample sizes were already large, with all experiments exceeding 50 repetitions, further repetitions would likely yield diminishing returns. As noted by Ghasemi and Zahediasl [122], when sample sizes exceed 30, deviations from normality typically do not present major statistical concerns.

8.5.5 Specific Energy Measurement Setup

As explained in Section 3.2, this thesis adopted a focused measurement scope targeting only containers directly related to the DYNAMOS data exchange archetypes. Containers that support monitoring (`prometheus`, `grafana`, `kepler`) and system-level Kubernetes components such as `coredns` were explicitly excluded from the energy consumption queries to isolate the impact of the experimental optimizations.

While this strategy ensures precision in measuring the effects of the optimizations, it also limits the generalizability of the reported energy usage to the entire system. For instance, the compression implementation might have influenced the energy consumption of networking components or monitoring tools—impacts that this study did not capture. Nevertheless, this exclusion was a deliberate scoping decision to ensure the results reflect only the energy footprint of the components directly involved in the focus of this thesis: execution of data exchange archetypes (see Section 3.2).

8.5.6 Additional Threats

We acknowledge that all experiments were conducted in only two environments—Docker Desktop with WSL2 and the FABRIC testbed—limiting the environmental diversity of the results. Although these setups provided a useful contrast between single-node and distributed Kubernetes deployments, results may differ on other cloud-native platforms or edge environments like AWS or Azure.

Furthermore, we did not evaluate potential impacts from external hardware constraints (e.g., CPU throttling, thermal effects), nor did we consider variability in energy pricing or carbon intensity. Energy pricing refers to differences in the cost of electricity across regions or providers, which may influence the financial implications of energy consumption. Carbon intensity, on the other hand, represents the amount of carbon dioxide (CO_2) emitted per unit of electricity consumed and varies depending on the underlying energy mix (e.g., coal, gas, renewables). These factors can significantly affect how energy efficiency is interpreted in real-world scenarios, especially in sustainability-driven or cost-sensitive deployments.

Despite these threats to validity, we believe that the methodological rigor, cross-environment testing, and statistical safeguards applied in this study yield valuable and credible insights into energy efficiency in data exchange systems.

8.6 Future Work

This chapter discussed the results of our research, provided answers to the research questions, highlighted additional insights, and addressed the limitations and threats to validity. To conclude, this section outlines potential directions for future work, identifying opportunities to expand and refine the research presented in this thesis.

8.6.1 Implementation & Validation of Additional Optimizations

This thesis identified and analyzed several energy efficiency optimizations applicable to data exchange archetypes (see Section 4.2). However, only a subset of these optimizations—specifically caching and compression—was implemented in DYNAMOS due to time constraints and the prioritization of deploying DYNAMOS in FABRIC. As a result, other promising optimizations, such as Query Execution Optimization (O3), were not included in the experimental evaluation.

Future research could incorporate these additional optimizations into DYNAMOS or similar systems to assess their impact on energy consumption. In particular, optimizing query execution has the potential to reduce processing time and energy usage, thereby improving overall system efficiency. Expanding the implementation scope would also help validate the broader applicability of these techniques and contribute to a deeper understanding of energy-efficient design in distributed data exchange systems.

8.6.2 Extending to Additional Archetypes

In this thesis, the focus was on a specific set of data exchange archetypes: Compute to Data and Data Through TTP. Future work could expand the scope by implementing optimizations across additional archetypes to evaluate their influence on energy consumption in a wider range of scenarios. For instance, the Prets use case[1] presents an opportunity for further exploration in DYNAMOS. In this use case, a machine learning model is collaboratively trained to predict building energy consumption using vertical federated learning (VFL). This approach enables multiple parties, each possessing different features of the same data subjects, to jointly train a model without sharing raw data. Investigating how energy efficiency optimizations impact such a system could provide additional valuable insights.

8.6.3 Applying Optimizations to Additional Data Exchange Systems

This thesis focused on implementing and validating energy efficiency optimizations in DYNAMOS. Future work could extend these optimizations to other data exchange systems to assess their effectiveness in different environments. By testing and refining these optimizations across multiple systems, researchers could determine whether similar energy efficiency gains are observed beyond DYNAMOS, strengthening the generalizability of the findings.

[1] https://www.prets.io/

8.6.4 Further Refinement of Optimizations

While Sections 5.1 and 5.2 detailed the implementations of caching and transferred data compression in DYNAMOS, further research could explore more advanced optimization techniques. Each of these optimizations could be refined and improved by:

- Comparing different caching strategies, such as in-memory caching versus distributed caching.
- Evaluating various compression algorithms and techniques to determine the most energy-efficient approach.
- Investigating hybrid approaches that combine multiple optimizations to maximize energy savings.

Future research could even dedicate an entire thesis to implementing and comparing these strategies in an effort to determine the most effective configurations for improving energy efficiency in data exchange systems.

8.6.5 Energy Consumption of Other Processes

As mentioned in Section 8.3.6, this thesis focused exclusively on measuring the energy consumption of the DYNAMOS data exchange archetypes and their associated containers. All queries were scoped to exclude unrelated infrastructure components such as `coredns`, `prometheus`, `grafana`, and `kepler`. While this approach allowed for targeted measurements of the optimizations studied in this thesis, it omitted the broader energy costs associated with the overall system infrastructure and monitoring setup. Future work could extend this research by quantifying the energy impact of these excluded components. This can be approached in two stages:

1. **Monitoring Overhead:** Measure the energy consumption of the monitoring stack itself, including `prometheus`, `grafana`, and `kepler`. These components enable energy transparency, but their own energy usage may be non-negligible, particularly in larger clusters. Understanding this cost is essential for assessing the net benefit of introducing energy monitoring tools.
2. **Kubernetes System Processes:** Measure the energy consumption of background Kubernetes services such as `coredns`, `kube-proxy`, and other control plane or network-related processes. These services are critical to cluster operation and contribute to baseline energy consumption, yet are often excluded in application-level evaluations.

Quantifying these additional energy sources would provide a more complete picture of the energy footprint of cloud-native/Kubernetes architectures. It would also help developers and infrastructure operators understand the full trade-offs involved when deploying observability tooling or when scaling their Kubernetes environments.

Chapter 9

Related Work

While conducting the research for this thesis, several related studies were identified across key topics. This chapter provides a brief overview of the main topics and their associated related work.

9.1 Energy Efficiency

Energy efficiency serves as the primary focus of this thesis. Foundational concepts of energy efficiency and green IT were explored through sources such as [2, 6, 12], which provided background and context. To investigate improvement techniques, comprehensive overviews from sources like [6, 13, 14, 66] were instrumental, offering insights and guiding further detailed research.

This deeper exploration led to specific topics such as energy code smells, refactoring techniques, and automated refactoring. Relevant sources include a significant number of research efforts [64, 65, 70, 79, 84]. Furthermore, research into energy measurement experiments informed the experimental framework for this thesis. Key sources like [18, 38, 78] offered valuable inspiration and practical guidance for structuring and conducting energy efficiency experiments.

9.2 Anomaly Detection & Root Cause Analysis of Microservices Energy Consumption

The paper by Floroiu *et al.* [1] is an important part of this thesis, providing an in-depth exploration of integrating Anomaly Detection (AD) and Root Cause Analysis (RCA) to analyze the energy consumption of microservices. This study served as the primary inspiration for incorporating the energy efficiency report pipeline into DYNAMOS. The paper offers both comprehensive theoretical insights and open-source practical examples, making it highly valuable for understanding how to introduce energy consumption measurements into microservice-based systems. For instance, it includes a detailed evaluation of suitable AD and RCA algorithms tailored to the energy efficiency report pipeline.

Techniques for collecting energy metrics were further examined through the tools documented in the study, such as [25] and [22], as well as an alternative tool, [27]. In addition, algorithms referenced in the paper, such as [34] and [36], were analyzed for their potential application in implementing the energy efficiency pipeline in DYNAMOS.

9.3 DYNAMOS: Dynamically Adaptive Microservice-based OS

The Master Software Engineering (SE) DYNAMOS thesis [5], the accompanying paper [9], and the open-source GitHub repository [107] are invaluable resources for this thesis, offering detailed insights into the design and implementation of the system. Furthermore, the thesis by Stutterheim [5] served as both an exemplary model and a source of inspiration for structuring this work. In his future research section, Stutterheim highlights that DYNAMOS has successfully demonstrated a framework and methodology for dynamically selecting data exchange archetypes. However, further research is needed into algorithms focused on 'green-IT.' This thesis aims to address this gap by contributing to the exploration of energy efficiency in data exchange systems, with a specific focus on DYNAMOS.

Chapter 10

Conclusion

This thesis explored the integration of energy efficiency optimizations into data exchange archetypes, with a specific focus on DYNAMOS, a dynamically adaptive microservice-based middleware. With the increasing demand for sustainable computing, optimizing energy consumption in microservice-based architectures has become a pressing challenge. Addressing this challenge required the identification of energy inefficiencies, followed by the selection and implementation of suitable optimizations.

Through a structured root cause analysis and literature review, we identified nine candidate optimizations applicable to data exchange systems. Two optimizations—caching requests and transferred data compression—were implemented and evaluated experimentally. Caching consistently reduced both energy consumption and execution time across all environments. In contrast, transferred data compression generally offered no significant energy savings and occasionally increased consumption due to compression and decompression overheads.

Beyond the primary results, this research revealed a moderate correlation between execution time and energy consumption, supporting the notion that performance improvements can enhance energy efficiency. However, this relationship proved weaker in the distributed FABRIC environment, underscoring the complexity of energy dynamics in multi-node systems and the need for direct energy measurements in such settings.

A key contribution and strength of this research lies in the deployment and validation of DYNAMOS in two fundamentally different environments: a local single-node Kubernetes setup using Docker Desktop with WSL2, and a distributed multi-node deployment on the FABRIC testbed. The FABRIC cluster was specifically configured to replicate real-world distributed conditions, enabling us to evaluate the portability and robustness of the implemented optimizations across environments. This dual-environment comparison revealed clear differences in orchestration latency, system stability, and energy monitoring granularity. Notably, the FABRIC environment—with its native Linux VMs—enabled more accurate and consistent energy measurements, including visibility into kernel-level processes that were obscured in the WSL2-based local setup. These findings confirm that full-system Linux environments provide superior monitoring fidelity and highlight the importance of infrastructure when interpreting energy efficiency results.

While this work demonstrated the feasibility of improving energy efficiency in DYNAMOS, several future research opportunities remain. These include exploring additional optimizations, quantifying the energy overhead of monitoring systems, and refining energy measurement methodologies in virtualized environments to enhance the accuracy of reported energy metrics.

Overall, this thesis contributes to the growing body of research on energy efficiency in software systems, demonstrating how targeted optimizations can enhance the sustainability of microservice-based architectures. By validating these findings in both local and distributed settings, this work supports broader global efforts toward sustainable computing, reinforcing the importance of energy-aware software design.

Acknowledgements

Throughout the writing of this thesis, I have received an incredible amount of support, guidance, and encouragement from many individuals, without whom this research would not have been possible. I would like to take this opportunity to express my deepest gratitude to all those who have contributed to this journey.

First and foremost, I am profoundly grateful to my supervisor, Dr. A.M. Oprescu, for her invaluable guidance and support throughout this thesis. Her insightful feedback and expertise have been instrumental in shaping both my research and my approach to scientific inquiry. Coming from a University of Applied Sciences background, I initially struggled to adapt to a research-oriented mindset. However, her patience and encouragement helped me bridge this gap, making me feel more confident in conducting rigorous academic research. Additionally, her guidance in statistics and mathematics, fields in which I had no prior background or education, played a crucial role in helping me make sense of my experimental results. Her dedication and mentorship have been a major driving force behind this thesis, and for that, I am truly grateful.

I would also like to extend my sincere appreciation to Jorrit Stutterheim for building DYNAMOS, the foundation of this research, and for taking the time during the summer of 2024 to help me familiarize myself with its architecture. His insights were invaluable in shaping my research direction, and his willingness to share his expertise made a significant impact on the early stages of my work.

A special thank you to Aleandro Mifsud, whose support and guidance throughout this thesis were invaluable. Alongside Jorrit, he played a key role in helping me formulate my research ideas during the summer. In January, he provided weekly feedback and served as a great source of inspiration for structuring my thesis. His assistance in navigating the DYNAMOS codebase, implementing optimizations, and debugging technical challenges, particularly in fixing the Data through TTP archetype, was immensely helpful. His enthusiasm, feedback, and technical expertise were instrumental in shaping the success of this research.

I would also like to thank Alexandros Koufakis for his valuable input in helping me refine my research ideas and for his support in implementing developments in DYNAMOS. His contributions were greatly appreciated throughout my thesis.

Furthermore, I am grateful to Georgia Samaritaki for her thoughtful feedback and guidance, which helped me refine my thesis further. I would also like to acknowledge Anestis Dalgkitsis for his assistance in understanding and working with FABRIC, which was an essential part of my research.

Additionally, I extend my thanks to my fellow students and friends, who engaged in discussions, offered valuable feedback or provided support in other ways. Their perspectives and insights played a crucial role in refining my ideas and strengthening my research approach.

Lastly, I am deeply appreciative of the unwavering support from my family and all others who have contributed to this journey, whether directly or indirectly. Their encouragement, patience, and motivation have been instrumental in keeping me focused and determined throughout this process.

To everyone who contributed to this thesis—whether through guidance, support, or encouragement—I sincerely appreciate your contributions and am deeply grateful.

Bibliography

[1] M. S. Floroiu, S. Russo, L. Giamattei, A. Guerriero, I. Malavolta, and R. Pietrantuono, "Anomaly detection and root cause analysis of microservices energy consumption," 2024, Vrije Universiteit Amsterdam, The Netherlands, Universita di Napoli Federico II, Italy, marXact B.V.

[2] C. Freitag, M. Berners-Lee, K. Widdicks, B. Knowles, G. S. Blair, and A. Friday, "The real climate and transformative impact of ict: A critique of estimates, trends, and regulations," *Patterns*, vol. 2, no. 9, p. 100 340, 2021, ISSN: 2666-3899. DOI: https://doi.org/10.1016/j.patter.2021.100340. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2666389921001884.

[3] J. Daniel, E. Guerra, T. Rosa, and A. Goldman, "Towards the detection of microservice patterns based on metrics," in *2023 49th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, 2023, pp. 132–139. DOI: 10.1109/SEAA60479.2023.00029.

[4] Y. Romani, O. Tibermacine, and C. Tibermacine, "Towards migrating legacy software systems to microservice-based architectures: A data-centric process for microservice identification," in *2022 IEEE 19th International Conference on Software Architecture Companion (ICSA-C)*, 2022, pp. 15–19. DOI: 10.1109/ICSA-C54293.2022.00010.

[5] J. Stutterheim, "Dynamos: Dynamically adaptive microservice-based os. a middleware for data exchange systems," M.S. thesis, University of Amsterdam, Amsterdam, The Netherlands, Jul. 2023, p. 82. [Online]. Available: https://ltvanbinsbergen.nl/files/theses/stutterheim-thesis.pdf.

[6] J. Balanza-Martinez, P. Lago, and R. Verdecchia, "Tactics for software energy efficiency: A review," in *Advances and New Trends in Environmental Informatics 2023*, V. Wohlgemuth, D. Kranzlmüller, and M. Höb, Eds., Cham: Springer Nature Switzerland, 2024, pp. 115–140.

[7] R. Wieringa and A. Morali, "Technical action research as a validation method in information systems design science," in *International Conference on Design Science Research in Information Systems and Technology*, Springer, 2012, pp. 220–238. [Online]. Available: https://api.semanticscholar.org/CorpusID:16131200.

[8] C. Guevara-Vega, B. Bernárdez, A. Durán, A. Quiña-Mera, M. Cruz, and A. Ruiz-Cortés, "Empirical strategies in software engineering research: A literature survey," in *2021 Second International Conference on Information Systems and Software Technologies (ICI2ST)*, 2021, pp. 120–127. DOI: 10.1109/ICI2ST51859.2021.00025.

[9] J. Stutterheim, A. Mifsud, and A. Oprescu, "Dynamos: Dynamic microservice composition for data-exchange systems, lessons learned," in *International Conference on Software Architecture (ICSA)*, Amsterdam, The Netherlands, Jun. 2024.

[10] S. Shakeri, L. Veen, and P. Grosso, "Evaluation of container overlays for secure data sharing," in *2020 IEEE 45th LCN Symposium on Emerging Topics in Networking (LCN Symposium)*, 2020, pp. 99–108. DOI: 10.1109/LCNSymposium50271.2020.9363266.

[11] E. Volynsky, M. Mehmed, and S. Krusche, "Architect: A framework for the migration to microservices," in *2022 International Conference on Computing, Electronics & Communications Engineering (iCCECE)*, 2022, pp. 71–76. DOI: 10.1109/iCCECE55162.2022.9875096.

[12] V. Agarwal, K. Sharma, and A. K. Rajpoot, "A review: Evolution of technology towards green it," in *2021 International Conference on Computing, Communication, and Intelligent Systems (ICCCIS)*, 2021, pp. 940–946. DOI: 10.1109/ICCCIS51004.2021.9397173.

[13] E. Capra, C. Francalanci, and S. A. Slaughter, "Measuring application software energy efficiency," *IT Professional*, vol. 14, no. 2, pp. 54–61, 2012. DOI: 10.1109/MITP.2012.39.

[14] L. Ardito, G. Procaccianti, M. Torchiano, and A. Vetrò, "Understanding green software development: A conceptual framework," *IT Professional*, vol. 17, no. 1, pp. 44–50, 2015. DOI: 10.1109/MITP.2015.16.

[15] GeeksforGeeks, *Difference between energy and power*, https://www.geeksforgeeks.org/difference-between-energy-and-power/, Accessed: 2025-01-22, 2024.

[16] M. Amaral *et al.*, "Process-based efficient power level exporter," in *2024 IEEE 17th International Conference on Cloud Computing (CLOUD)*, 2024, pp. 456–467. DOI: 10.1109/CLOUD62652.2024.00058.

[17] M. Amaral *et al.*, "Kepler: A framework to calculate the energy consumption of containerized applications," in *2023 IEEE 16th International Conference on Cloud Computing (CLOUD)*, 2023, pp. 69–71. DOI: 10.1109/CLOUD60044.2023.00017.

[18] L. Koedijk and A. Oprescu, "Finding significant differences in the energy consumption when comparing programming languages and programs," in *2022 International Conference on ICT for Sustainability (ICT4S)*, 2022, pp. 1–12. DOI: 10.1109/ICT4S55073.2022.00012.

[19] Cloud Native Computing Foundation, *Kepler*, Accessed: 2025-01-22, 2024. [Online]. Available: https://sustainable-computing.io/usage/deep_dive/.

[20] Python.org, *The python tutorial*, https://docs.python.org/3/tutorial/index.html, Accessed: 2024-06-24, 2024.

[21] R. Sedgewick and K. Wayne, *Algorithms, 4th Edition*. Addison-Wesley Professional, 2011, p. 4, https://www.oreilly.com/library/view/algorithms-4th-edition/9780132762564/, ISBN: 9780132762564.

[22] Prometheus, *Prometheus documentation*, Accessed: 2024-06-24, 2024. [Online]. Available: https://prometheus.io/docs/introduction/overview/.

[23] Kubernetes, *Kubernetes overview*, https://kubernetes.io/docs/concepts/overview/, Accessed: 2024-06-24, 2024.

[24] GitHub, *The 10 best tools to green your software*, https://github.blog/open-source/social-impact/the-10-best-tools-to-green-your-software, Accessed: 2024-12-20, 2023.

[25] Hubblo-org, *Scaphandre*, Accessed: 2024-06-24, 2024. [Online]. Available: https://hubblo-org.github.io/scaphandre-documentation/index.html.

[26] eBPF Project, *What is ebpf?* https://ebpf.io/what-is-ebpf/, Accessed: 2025-01-06, 2025.

[27] Cloud Native Computing Foundation, *Kepler*, Accessed: 2024-07-08, 2024. [Online]. Available: https://sustainable-computing.io/.

[28] Cloud Native Computing Foundation. "Exploring kepler's potentials: Unveiling cloud application power consumption." Accessed: 2024-12-13. (Oct. 2023), [Online]. Available: https://www.cncf.io/blog/2023/10/11/exploring-keplers-potentials-unveiling-cloud-application-power-consumption/.

[29] K. L. Huamin Chen Parul Singh. "Sustainability, the cloud native way." Accessed: 2024-12-17. (Feb. 2023), [Online]. Available: https://next.redhat.com/2023/02/21/sustainability-the-cloud-native-way/.

[30] H. C. Parul Singh. "Introducing kepler: Efficient power monitoring for kubernetes." Accessed: 2024-12-17. (Aug. 2023), [Online]. Available: https://next.redhat.com/2023/08/22/introducing-kepler-efficient-power-monitoring-for-kubernetes/.

[31] M. Akbari, R. Bolla, R. Bruschi, F. Davoli, C. Lombardo, and B. Siccardi, "A monitoring, observability and analytics framework to improve the sustainability of b5g technologies," in *2024 IEEE International Conference on Communications Workshops (ICC Workshops)*, 2024, pp. 969–975. DOI: 10.1109/ICCWorkshops59551.2024.10615948.

[32] C. Centofanti, J. Santos, V. Gudepu, and K. Kondepu, "Impact of power consumption in containerized clouds: A comprehensive analysis of open-source power measurement tools," *Computer Networks*, vol. 245, p. 110 371, 2024.

[33] V. Gudepu, R. R. Tella, C. Centofanti, J. Santos, A. Marotta, and K. Kondepu, "Demonstrating the energy consumption of radio access networks in container clouds," in *NOMS2024, the IEEE/IFIP Network Operations and Management Symposium*, 2024.

[34] D. Wang, M. Nie, and D. Chen, "Bae: Anomaly detection algorithm based on clustering and autoencoder," *Mathematics*, vol. 11, no. 15, 2023, ISSN: 2227-7390. DOI: 10.3390/math11153398. [Online]. Available: https://www.mdpi.com/2227-7390/11/15/3398.

[35] T. Zhang, R. Ramakrishnan, and M. Livny, "Birch: An efficient data clustering method for very large databases," *SIGMOD Rec.*, vol. 25, no. 2, pp. 103–114, Jun. 1996, ISSN: 0163-5808. DOI: 10.1145/235968.233324. [Online]. Available: https://doi.org/10.1145/235968.233324.

[36] A. Ikram, S. Chakraborty, S. Mitra, S. Saini, S. Bagchi, and M. Kocaoglu, "Root cause analysis of failures in microservices through causal discovery," in *Advances in Neural Information Processing Systems*, S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, Eds., vol. 35, Curran Associates, Inc., 2022, pp. 31 158–31 170. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2022/file/c9fcd02e6445c7dfbad6986abee53d0d-Paper-Conference.pdf.

[37] Microsoft Learn, *What is the windows subsystem for linux?* Accessed: 2025-01-23, 2023. [Online]. Available: https://learn.microsoft.com/en-us/windows/wsl/about.

[38] L. Cruz, *Green software engineering done right: A scientific guide to set up energy efficiency experiments*, http://luiscruz.github.io/2021/10/10/scientific-guide.html, Blog post., 2021. DOI: 10.6084/m9.figshare.22067846.v1.

[39] O. S. Daniel Kahneman and C. R. Sunstein, *Noise: A Flaw in Human Judgment*. New York, USA: Little, Brown Spark, 2021.

[40] LINKERD, *Linkerd architecture*, https://linkerd.io/2.15/reference/architecture/, Accessed: 2025-01-07, 2025.

[41] GeeksforGeeks, *What is linkerd?* https://www.geeksforgeeks.org/what-is-linkerd/, Accessed: 2025-01-07, 2024.

[42] LINKERD, *What is a service mesh?* https://linkerd.io/what-is-a-service-mesh/, Accessed: 2025-01-07, 2025.

[43] Kubernetes, *Ingress*, https://kubernetes.io/docs/concepts/services-networking/ingress/, Accessed: 2025-01-07, 2025.

[44] T. Yuki and S. Rajopadhye, "Folklore confirmed: Compiling for speed compiling for energy," in *International Workshop on Languages and Compilers for Parallel Computing*, Springer, 2013, pp. 169–184.

[45] S. Vos, P. Lago, R. Verdecchia, and I. Heitlager, "Architectural tactics to optimize software for energy efficiency in the public cloud," in *2022 International Conference on ICT for Sustainability (ICT4S)*, 2022, pp. 77–87. DOI: 10.1109/ICT4S55073.2022.00019.

[46] Z. Wang, Y. Wen, J. Chen, B. Cao, and F. Wang, "Towards energy-efficient scheduling with batch processing for instance-intensive cloud workflows," in *2018 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Ubiquitous Computing & Communications, Big Data & Cloud Computing, Social Computing & Networking, Sustainable Computing & Communications (ISPA/IUCC/BDCloud/SocialCom/SustainCom)*, 2018, pp. 590–596. DOI: 10.1109/BDCloud.2018.00092.

[47] G. Pinto, F. Castor, and Y. D. Liu, "Mining questions about software energy consumption," in *Proceedings of the 11th working conference on mining software repositories*, 2014, pp. 22–31.

[48] GeeksforGeeks, *Caching – system design concept*, https://www.geeksforgeeks.org/caching-system-design-concept-for-beginners/, Accessed: 2025-01-10, 2024.

[49] L. B. Mohammed, A. Anpalagan, and M. Jaseemuddin, "Energy and latency efficient caching in mobile edge networks: Survey, solutions, and challenges," *Wireless Personal Communications*, vol. 129, no. 2, pp. 1249–1283, 2023.

[50] H. Xu, H. Huang, and G. Wang, "An energy-efficient caching strategy based on coordinated caching for green content-centric network," in *Security, Privacy and Anonymity in Computation, Communication and Storage*, G. Wang, I. Ray, J. M. Alcaraz Calero, and S. M. Thampi, Eds., Cham: Springer International Publishing, 2016, pp. 168–175.

[51] GeeksforGeeks, *Introduction to data compression*, https://www.geeksforgeeks.org/introduction-to-data-compression/, Accessed: 2025-01-10, 2021.

[52] B. R. Stojkoska and Z. Nikolovski, "Data compression for energy efficient iot solutions," in *2017 25th Telecommunication Forum (TELFOR)*, 2017, pp. 1–4. DOI: 10.1109/TELFOR.2017.8249368.

[53] M. Hajiloo Vakil and Z. Shirmohammadi, "Edc-er: An efficient data compression method for energy reduction in wbans," *IEEE Access*, vol. 12, pp. 155 274–155 286, 2024. DOI: 10.1109/ACCESS.2024.3476424.

[54] H. M. Al-Kadhim and H. S. Al-Raweshidy, "Energy efficient data compression in cloud based iot," *IEEE Sensors Journal*, vol. 21, no. 10, pp. 12 212–12 219, 2021. DOI: 10.1109/JSEN.2021.3064611.

[55] K. T. M. Tran, A. X. Pham, N. P. Nguyen, and P. T. Dang, "Analysis and performance comparison of iot message transfer protocols applying in real photovoltaic system," *International Journal of Networked and Distributed Computing*, pp. 1–13, 2024.

[56] N. Naik, "Choice of effective messaging protocols for iot systems: Mqtt, coap, amqp and http," in *2017 IEEE international systems engineering symposium (ISSE)*, IEEE, 2017, pp. 1–7.

[57] F. Z. Chafi, Y. Fakhri, and F. Z. A. H. Aadi, "Introduction to internet of things' communication protocols," in *Advanced Intelligent Systems for Sustainable Development (AI2SD'2020)*, J. Kacprzyk, V. E. Balas, and M. Ezziyyani, Eds., Cham: Springer International Publishing, 2022, pp. 142–150.

[58] L. Kamiński, M. Kozłowski, D. Sporysz, K. Wolska, P. Zaniewski, and R. Roszczyk, "Comparative review of selected internet communication protocols," *Foundations of Computing and Decision Sciences*, vol. 48, no. 1, pp. 39–56, 2023. DOI: doi:10.2478/fcds-2023-0003. [Online]. Available: https://doi.org/10.2478/fcds-2023-0003.

[59] C. L. Chamas, D. Cordeiro, and M. M. Eler, "Comparing rest, soap, socket and grpc in computation offloading of mobile applications: An energy cost analysis," in *2017 IEEE 9th Latin-American Conference on Communications (LATINCOM)*, 2017, pp. 1–6. DOI: 10.1109/LATINCOM.2017.8240185.

[60] R. Pereira *et al.*, "Energy efficiency across programming languages: How do energy, time, and memory relate?" In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*, ser. SLE 2017, Vancouver, BC, Canada: Association for Computing Machinery, 2017, pp. 256–267, ISBN: 9781450355254. DOI: 10.1145/3136014.3136031. [Online]. Available: https://doi.org/10.1145/3136014.3136031.

[61] R. Pereira *et al.*, "Ranking programming languages by energy efficiency," *Science of Computer Programming*, vol. 205, p. 102 609, 2021, ISSN: 0167-6423. DOI: https://doi.org/10.1016/j.scico.2021.102609. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167642321000022.

[62] S. Abdulsalam, D. Lakomski, Q. Gu, T. Jin, and Z. Zong, "Program energy efficiency: The impact of language, compiler and implementation choices," in *International Green Computing Conference*, 2014, pp. 1–6. DOI: 10.1109/IGCC.2014.7039169.

[63] M. Couto, R. Pereira, F. Ribeiro, R. Rua, and J. Saraiva, "Towards a green ranking for programming languages," in *Proceedings of the 21st Brazilian Symposium on Programming Languages*, ser. SBLP '17, Fortaleza, CE, Brazil: Association for Computing Machinery, 2017, ISBN: 9781450353892. DOI: 10.1145/3125374.3125382. [Online]. Available: https://doi.org/10.1145/3125374.3125382.

[64] A. Vetrò, L. Ardito, G. Procaccianti, and M. Morisio, "Definition, implementation and validation of energy code smells: An exploratory study on an embedded system," *Energy*, pp. 34–39, 2013. [Online]. Available: https://api.semanticscholar.org/CorpusID:54836561.

[65] A. Imran, T. Kosar, J. Zola, and M. F. Bulut, "Towards sustainable cloud software systems through energy-aware code smell refactoring," in *2024 IEEE 17th International Conference on Cloud Computing (CLOUD)*, 2024, pp. 223–234. DOI: 10.1109/CLOUD62652.2024.00034.

[66] M. Ileana, "Optimizing energy efficiency in distributed web systems," in *2023 7th International Symposium on Innovative Approaches in Smart Technologies (ISAS)*, 2023, pp. 1–5. DOI: 10.1109/ISAS60782.2023.10391617.

[67] F. Palomba, D. Di Nucci, A. Panichella, A. Zaidman, and A. De Lucia, "On the impact of code smells on the energy consumption of mobile applications," *Information and Software Technology*, vol. 105, pp. 43–55, 2019, ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2018.08.004. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584918301678.

[68] H. Anwar, D. Pfahl, and S. N. Srirama, "Evaluating the impact of code smell refactoring on the energy consumption of android applications," in *2019 45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, 2019, pp. 82–86. DOI: 10.1109/SEAA.2019.00021.

[69] Reeshti, R. Sehgal, D. Mehrotra, R. Nagpal, and T. Choudhury, "Code smell refactoring for energy optimization of android apps," in *Innovations in Cyber Physical Systems: Select Proceedings of ICICPS 2020*, Springer, 2021, pp. 371–379.

[70] S. Georgiou, S. Rizou, and D. Spinellis, "Software development lifecycle for energy efficiency: Techniques and tools," *ACM Comput. Surv.*, vol. 52, no. 4, Aug. 2019, ISSN: 0360-0300. DOI: 10.1145/3337773. [Online]. Available: https://doi.org/10.1145/3337773.

[71] C. Fu, D. Qian, T. Huang, and Z. Luan, "Code-level optimization for program energy consumption," in *The Tenth International Conference on Computational Logics, Algebras, Programming, Tools, and Benchmarking (COMPUTATION TOOLS 2019)*, IARIA, 2019, pp. 1–4, ISBN: 978-1-61208-709-2.

[72] A. A. Alsayyah and S. Ahmed, "Energy efficient software development techniques for cloud based applications," *International Journal*, vol. 9, no. 5, 2020.

[73] İ. Şanlıalp, M. M. Öztürk, and T. Yiğit, "Energy efficiency analysis of code refactoring techniques for green and sustainable software in portable devices," *Electronics*, vol. 11, no. 3, 2022, ISSN: 2079-9292. [Online]. Available: https://www.mdpi.com/2079-9292/11/3/442.

[74] J. Corral-García, F. Lemus-Prieto, J.-L. González-Sánchez, and M.-Á. Pérez-Toledano, "Analysis of energy consumption and optimization techniques for writing energy-efficient code," *Electronics*, vol. 8, no. 10, 2019, ISSN: 2079-9292. DOI: 10.3390/electronics8101192. [Online]. Available: https://www.mdpi.com/2079-9292/8/10/1192.

[75] M. Gottschalk, J. Jelschen, and A. Winter, "Saving energy on mobile devices by refactoring.," in *EnviroInfo*, Citeseer, 2014, pp. 437–444.

[76] A. Noureddine and A. Rajan, "Optimising energy consumption of design patterns," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 2, 2015, pp. 623–626. DOI: 10.1109/ICSE.2015.208.

[77] S. van Oostveen, "Regarding the impact of code smells on the energy efficiency of different computer languages," bachelorthesis, Universiteit van Amsterdam, Jun. 2020.

[78] P. Sommerhalter, "Calabash: An analysis framework and catalog for green code patterns in software," masterthesis, Universiteit van Amsterdam, Aug. 2024.

[79] D. Connolly Bree and M. Ó. Cinnéide, "Automated refactoring for energy-aware software," in *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2021, pp. 689–694. DOI: 10.1109/ICSME52107.2021.00082.

[80] Sonar, *Sonarqube 10.7 documentation*, https://docs.sonarsource.com/sonarqube/latest/, Accessed: 2024-10-23, 2024.

[81] A. Ribeiro, J. F. Ferreira, and A. Mendes, "Ecoandroid: An android studio plugin for developing energy-efficient java mobile applications," in *2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)*, 2021, pp. 62–69. DOI: 10.1109/QRS54544.2021.00017.

[82] R. Morales, R. Saborido, F. Khomh, F. Chicano, and G. Antoniol, "Earmo: An energy-aware refactoring approach for mobile apps," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, 2018, pp. 59–59. DOI: 10.1145/3180155.3182524.

[83] L. Cruz, R. Abreu, and J.-N. Rouvignac, "Leafactor: Improving energy efficiency of android apps via automatic refactoring," in *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, 2017, pp. 205–206. DOI: 10.1109/MOBILESoft.2017.21.

[84] E. Iannone, F. Pecorelli, D. Di Nucci, F. Palomba, and A. De Lucia, "Refactoring android-specific energy smells: A plugin for android studio," in *2020 IEEE/ACM 28th International Conference on Program Comprehension (ICPC)*, 2020, pp. 451–455. DOI: 10.1145/3387904.3389298.

[85] G. Procaccianti, H. Fernández, and P. Lago, "Empirical evaluation of two best practices for energy-efficient software development," *Journal of Systems and Software*, vol. 117, pp. 185–198, 2016, ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2016.02.035. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121216000777.

[86] GeeksforGeeks, *What are in-memory caches?* https://www.geeksforgeeks.org/what-are-in-memory-caches/, Accessed: 2025-01-14, 2024.

[87] Baeldung, *Memcached vs redis*, https://www.baeldung.com/memcached-vs-redis, Accessed: 2025-01-14, 2024.

[88] GeeksforGeeks, *Difference between redis and memcached*, https://www.geeksforgeeks.org/difference-between-redis-and-memcached/, Accessed: 2025-01-14, 2024.

[89] GeeksforGeeks, *Memcached vs. redis*, https://www.geeksforgeeks.org/memcached-vs-redis/, Accessed: 2025-01-14, 2024.

[90] GeeksforGeeks, *Why caching does not always improve performance?* https://www.geeksforgeeks.org/why-caching-does-not-always-improve-performance/, Accessed: 2025-01-14, 2024.

[91] WMTIPS, *Compression - most popular*, https://www.wmtips.com/technologies/compression/, Accessed: 2025-01-21, 2025.

[92] W3Techs, *Usage statistics of compression for websites*, https://w3techs.com/technologies/details/ce-compression, Accessed: 2025-01-21, 2025.

[93] K. Kryukov, M. T. Ueda, S. Nakagawa, and T. Imanishi, "Sequence compression benchmark (scb) database—a comprehensive evaluation of reference-free compressors for fasta-formatted sequences," *GigaScience*, vol. 9, no. 7, giaa072, Jul. 2020, ISSN: 2047-217X. DOI: 10.1093/gigascience/giaa072. eprint: https://academic.oup.com/gigascience/article-pdf/9/7/giaa072/60689358/gigascience_9_7_giaa072_s14.pdf. [Online]. Available: https://doi.org/10.1093/gigascience/giaa072.

[94] Z. Syed and T. Soomro, "Compression algorithms: Brotli, gzip and zopfli perspective," *Indian Journal of Science and Technology*, vol. 11, no. 45, pp. 1–4, 2018.

[95] FABRIC, *Fabric*, Accessed: 2025-03-31, 2025. [Online]. Available: https://portal.fabric-testbed.net/.

[96] I. Baldin *et al.*, "Fabric: A national-scale programmable experimental network infrastructure," *IEEE Internet Computing*, vol. 23, no. 6, pp. 38–47, 2019. DOI: 10.1109/MIC.2019.2958545.

[97] L. Bryant, R. W. Gardner, F. Hu, D. Jordan, and R. P. Taylor, "Kubernetes deployment options for on-prem clusters," *arXiv preprint arXiv:2407.01620*, 2024.

[98] C. G. Sophie Turol and S. Matykevich, *A multitude of kubernetes deployment tools: Kubespray, kops, and kubeadm*, Accessed: 2025-03-31, 2018. [Online]. Available: https://www.altoros.com/blog/a-multitude-of-kubernetes-deployment-tools-kubespray-kops-and-kubeadm/.

[99] R. R. Singh, *Kubernetes deployment - which tool?* Accessed: 2025-03-31, 2019. [Online]. Available: https://faun.pub/kubernetes-deployment-which-tool-7e6eaca99dfa.

[100] F. Hu, *Service deployment in fabric at cern*, Accessed: 2025-03-31, 2023. [Online]. Available: https://agenda.hep.wisc.edu/event/2014/contributions/28504/attachments/9175/11071/2023.07.12%20Service%20Deployment%20in%20FABRIC%20at%20CERN.pdf.

[101] O. Nath, *Single cluster vs. multiple clusters: How many should you have in a kubernetes deployment?* Accessed: 2025-03-31, 2022. [Online]. Available: https://www.spiceworks.com/tech/devops/articles/single-or-multiple-clusters-for-kubernetes-deployment/.

[102] M. Perry, *Kubernetes multi-cluster: Why and when to use them*, Accessed: 2025-03-31, 2022. [Online]. Available: https://www.qovery.com/blog/kubernetes-multi-cluster-why-and-when-to-use-them/.

[103] S. Boydadaev, *What kubernetes network plugin should you use? a side by side comparison*, Accessed: 2025-04-11, 2024. [Online]. Available: https://hackernoon.com/what-kubernetes-network-plugin-should-you-use-a-side-by-side-comparison.

[104] R. Kumar and M. C. Trivedi, "Networking analysis and performance comparison of kubernetes cni plugins," in *Advances in Computer, Communication and Computational Sciences: Proceedings of IC4S 2019*, Springer, 2021, pp. 99–109.

[105] N. Kapočius, "Performance studies of kubernetes network solutions," in *2020 IEEE Open Conference of Electrical, Electronic and Information Sciences (eStream)*, 2020, pp. 1–6. DOI: 10.1109/eStream50540.2020.9108894.

[106] Kubernetes, *Kubernetes components*, Accessed: 2025-04-14, 2024. [Online]. Available: https://kubernetes.io/docs/concepts/overview/components/.

[107] J. Stutterheim, *Dynamos github*, Accessed: 2025-03-31, 2024. [Online]. Available: https://github.com/Jorrit05/DYNAMOS.

[108] N. M. Razali, Y. B. Wah, *et al.*, "Power comparisons of shapiro-wilk, kolmogorov-smirnov, lilliefors and anderson-darling tests," *Journal of statistical modeling and analytics*, vol. 2, no. 1, pp. 21–33, 2011.

[109] P. Mishra, C. M. Pandey, U. Singh, A. Gupta, C. Sahu, and A. Keshri, "Descriptive statistics and normality tests for statistical data," *Annals of cardiac anaesthesia*, vol. 22, no. 1, pp. 67–72, 2019.

[110] N. Nachar *et al.*, "The mann-whitney u: A test for assessing whether two independent samples come from the same distribution," *Tutorials in quantitative Methods for Psychology*, vol. 4, no. 1, pp. 13–20, 2008.

[111] H. W. Wendt, "Dealing with a common problem in social science: A simplified rank-biserial coefficient of correlation based on the u statistic.," *European Journal of Social Psychology*, vol. 2, no. 4, 1972.

[112] B. Fruchter and J. P. Guilford, *Fundamental statistics in psychology and education*. McGraw-Hill, 1983.

[113] E. S. Pearson, "Some notes on sampling tests with two variables," *Biometrika*, pp. 337–360, 1929.

[114] M. G. Kendall, "A new measure of rank correlation," *Biometrika*, vol. 30, no. 1-2, pp. 81–93, 1938.

[115] S. S. Malleni, R. S. Canavate, and V. Challa, "Into the fire: Delving into kubernetes performance and scale with kube-burner," in *Companion of the 15th ACM/SPEC International Conference on Performance Engineering*, 2024, pp. 89–90.

[116] I. M. Ramadan, C. Centofanti, A. Marotta, and F. Graziosi, "Evaluating kubernetes distributions: Insights from stress testing scenarios," in *2025 17th International Conference on COMmunication Systems and NETworks (COMSNETS)*, 2025, pp. 13–18. DOI: 10.1109/COMSNETS63942.2025.10885637.

[117] R. Verdecchia, G. Procaccianti, I. Malavolta, P. Lago, and J. Koedijk, "Estimating energy impact of software releases and deployment strategies: The kpmg case study," in *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2017, pp. 257–266. DOI: 10.1109/ESEM.2017.39.

[118] H. Acar, G. I. Alptekin, J.-P. Gelas, and P. Ghodous, "The impact of source code in software on power consumption," *International Journal of Electronic Business Management*, vol. 14, pp. 42–52, 2016.

[119] L. G. Lima, F. Soares-Neto, P. Lieuthier, F. Castor, G. Melfe, and J. P. Fernandes, "Haskell in green land: Analyzing the energy behavior of a purely functional language," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1, 2016, pp. 517–528. DOI: 10.1109/SANER.2016.85.

[120] G. Pinto, F. Castor, and Y. D. Liu, "Understanding energy behaviors of thread management constructs," in *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, 2014, pp. 345–360.

[121] A. E. Trefethen and J. Thiyagalingam, "Energy-aware software: Challenges, opportunities and strategies," *Journal of Computational Science*, vol. 4, no. 6, pp. 444–449, 2013, Scalable Algorithms for Large-Scale Systems Workshop (ScalA2011), Supercomputing 2011, ISSN: 1877-7503. DOI: https://doi.org/10.1016/j.jocs.2013.01.005. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877750313000173.

[122] A. Ghasemi and S. Zahediasl, "Normality tests for statistical analysis: A guide for non-statisticians," *International Journal of Endocrinology and Metabolism*, vol. 10, no. 2, pp. 486–489, DOI: 10.5812/ijem.3505. [Online]. Available: https://brieflands.com/articles/ijem-71904.