POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Ermes: enabling Stateful Serverless at the Edge through a locality-aware Migration Policy

Tesi di Laurea Magistrale in
Computer Science and Engineering e High Performance Computing Engineering - Ingegneria Informatica ed Ingegneria del Calcolo ad Alte Prestazioni

Authors: **Matteo Briscini, Matteo Cenzato**

Student IDs: 259098, 245565
Advisor: Prof. Alessandro Margara
Co-advisors: Dario d'Abate, Arianna Dragoni
Academic Year: 2024-25

Abstract

Serverless computing has emerged as a prominent paradigm for deploying applications without the overhead of underlying infrastructure management. Initially restricted to cloud environments, this model is rapidly expanding toward the network edge to accommodate latency-sensitive applications. Furthermore, its scope is expanding towards stateful environments, as modern workloads increasingly require access to data stores to maintain application state. Despite various efforts to adapt serverless computing for edge and stateful applications, current solutions lack a comprehensive Function-as-a-Service (FaaS) architecture that integrates internal storage, transparent migration for user locality, and data-aware function invocation.

In this thesis we present Ermes, a platform designed to support stateful applications within the edge-cloud continuum. Our contributions are twofold. First, we establish a theoretical foundation for optimal resource placement by formulating a centralized Integer Linear Programming (ILP) model that minimizes global response times under node storage and computational constraints. Second, we propose a decentralized migration policy inspired by mechanical potential energy, where nodes autonomously move state and logic toward higher-demand areas of the network. Finally, we implement this framework into a functional distributed platform, providing a lightweight execution environment optimized for resource-constrained edge nodes.

We evaluate our design by first assessing the computational limits of the centralized ILP model, used to establish a theoretical baseline. We then demonstrate that our distributed policy, integrated within the Ermes platform, closely aligns with this mathematical optimum. Furthermore, we conduct extensive stress tests to evaluate platform scalability under high request volumes, perform an ablation study on individual system components, and validate the platform's effectiveness in maintaining low latency during client migration. Our results indicate that Ermes effectively bridges the gap between serverless and edge computing, providing seamless data-aware execution in dynamic environments.

Keywords: Serverless Computing, Edge Computing, Stateful, Data Placement, Optimization, Distributed Systems

Abstract in lingua italiana

Il serverless computing si è affermato come paradigma di rilievo per l'implementazione di applicazioni senza l'onere della gestione dell'infrastruttura sottostante. Inizialmente limitato agli ambienti cloud, questo modello si sta rapidamente espandendo verso l'edge della rete per supportare applicazioni sensibili alla latenza. Inoltre, il suo ambito si sta estendendo verso applicazioni stateful, poiché i carichi di lavoro moderni richiedono sempre più l'accesso ad archivi di dati per mantenere lo stato delle applicazioni. Nonostante i vari sforzi per adattare il serverless computing all'edge e alle applicazioni stateful, le soluzioni attuali mancano di un'architettura completa Function-as-a-Service (FaaS) che integri storage interno, migrazione trasparente in base alla posizione dell'utente e invocazione di funzioni basate sui dati.

In questa tesi, presentiamo Ermes, una piattaforma progettata per supportare applicazioni stateful nel continuum edge-cloud. Il nostro contributo è duplice. In primo luogo, stabiliamo una base teorica per il posizionamento ottimale delle risorse formulando un modello centralizzato di Programmazione Lineare Intera (ILP) che riduce al minimo i tempi di risposta globali in base ai vincoli di storage e computazionali dei nodi. In secondo luogo, proponiamo una politica di migrazione decentralizzata ispirata all'energia potenziale meccanica, in cui i nodi spostano autonomamente stato e logica verso aree della rete a maggiore richiesta. Infine, implementiamo questo framework in una piattaforma distribuita funzionale, fornendo un ambiente di esecuzione leggero e ottimizzato per nodi edge con risorse limitate.

Valutiamo il nostro sistema prima mostrando limiti computazionali del modello ILP centralizzato, usato per stabilire una base teorica. Dimostriamo quindi che la nostra politica distribuita, integrata nella piattaforma Ermes, si allinea strettamente con questo optimum matematico. Inoltre, conduciamo approfonditi stress test per valutare la scalabilità della piattaforma in presenza di elevati volumi di richieste, eseguiamo uno studio di ablazione sui singoli componenti del sistema e convalidiamo l'efficacia della piattaforma nel mantenere una bassa latenza durante la migrazione dei client. I nostri risultati indicano che Ermes colma efficacemente il divario tra serverless ed edge computing, fornendo un'esecuzione fluida e basata sui dati in ambienti dinamici.

Parole chiave: Computazione Serverless, Computazione all'Edge, Stateful, Posizionamento dei dati, Ottimizzazione, Sistemi Distribuiti

Contents

1 | Introduction

The paradigm shift toward *Cloud Computing* and its subsequent evolution into *Serverless Computing* has revolutionized software development by abstracting infrastructure management. By allowing developers to deploy code as modular functions, these models offer unprecedented elasticity and operational simplicity. However, the architectural design of current serverless platforms is fundamentally rooted in a *stateless* execution model. In this model, internal state is not preserved between invocations, requiring applications to rely on external, often remote, storage for persistence.

This characteristic imposes significant limitations on modern applications, particularly those involving IoT and Big Data. For instance, IoT deployments often require real-time processing of high-frequency telemetry data, where the overhead of establishing a new connection to an external database for every invocation introduces significant latency [35]. Similarly, Big Data workflows involving complex state transitions or iterative processing are hindered by the 'data shipping' problem, the constant movement of large datasets between ephemeral execution environments and remote storage [41].

To overcome this, there is a growing need to leverage *Edge Computing*, a paradigm that places computational resources and storage in the physical proximity of data sources, such as gateways or local micro-datacenters.

In this thesis, we propose **Ermes**, a novel platform that unifies serverless and edge computing through a distributed, state-aware architecture. Our work focuses on solving the data-locality problem by integrating storage and execution into a single, cohesive framework.

The core of our research is a dual-policy approach to data placement. We first develop a *centralized optimization model* that serves as a theoretical basis, allowing us to identify the absolute optimal placement of functions and data. Recognizing the scalability limits of centralized solvers in large-scale networks, we then propose a *decentralized migration policy* inspired by stable mechanical equilibrium. In a physical system, an object moves spontaneously if that movement decreases its mechanical potential energy. Similarly, Ermes nodes autonomously move state toward higher-demand areas of the network. This

ensures that the system self-organizes to minimize the distance between the user and the required state without requiring a global orchestrator.

To validate these concepts, we implemented the Ermes platform using *WebAssembly* (WASM) as a lightweight execution format to enable rapid deployment at the edge and conducted an extensive experimental campaign. Our evaluation demonstrates the synergy between our theoretical and practical contributions: we use the centralized model to benchmark the effectiveness of our distributed policy, proving that our decentralized approach reaches near-optimal configurations. Finally, we show that the platform remains robust under heavy workloads and transparently handles client mobility, significantly reducing user-side latency compared to traditional cloud-centric FaaS.

The remainder of this thesis is organized as follows:

- **Chapters 2 and 3** explore the state of the art, highlighting the characteristics of current technologies and research directions while identifying the gaps that prevent the realization of a complete stateful serverless system.
- **Chapter 4** describes our proposed solution, providing a theoretical overview of the system topology, replication strategies, and the communication protocols required for a seamless implementation.
- **Chapters 5 and 6** formalize the data placement problem, presenting both the centralized mathematical baseline and the distributed physical-inspired policy.
- **Chapter 7** describes the implementation of the proposed system. We focus on the realization of the distributed data placement policy and analyze the foundational technologies utilized across the various architectural components.
- **Chapter 8** presents a comprehensive evaluation. We analyze the computational limits of the centralized solver, measure the optimality gap of the distributed policy, and assess the platform's performance regarding scalability, component overhead, and resilience to client migration.
- **Chapter 9** summarizes our findings, discusses the limitations of the proposed model, and outlines potential future research directions in stateful serverless computing.

2 | Background & Motivation

This chapter provides an overview of the fundamental characteristics of Serverless and Edge Computing, examining the challenges associated with unifying these two paradigms. Specifically, we discuss how the integration of state management within the computing continuum can expand the scope of traditional application development. Furthermore, we illustrate how our proposed platform, Ermes, integrates into diverse real-world scenarios to address these challenges.

2.1. Serverless Computing

The evolution of *Cloud Computing* has introduced diverse programming paradigms designed to abstract underlying infrastructure. Early offerings, such as Amazon EC2, provided low-level virtual machines (VMs) that allowed users to replicate local environments in the cloud. However, these systems required developers to manage VMs directly, including complex configuration and resource provisioning.

To address these operational complexities, the industry shifted toward *Serverless Computing* [31]. In this paradigm, developers write modular functions and upload them to a Function-as-a-Service (FaaS) environment. The cloud provider assumes full responsibility for maintenance, scaling, and provisioning, often complemented by Backend-as-a-Service (BaaS) offerings [29]. These functions are typically invoked in an event-driven manner, providing scalable, on-demand resources that eliminate the need for significant upfront infrastructure investment.

The defining characteristics of traditional serverless platforms include:

- **Infrastructure Abstraction:** The platform manages the physical and virtual hardware, allowing developers to focus exclusively on core application logic.
- **Elasticity:** The platform continuously monitors runtime resource utilization and dynamically scales the application based on the volume of incoming requests.
- **Pay-per-Use Billing:** Costs are incurred solely based on the resources consumed

during execution, eliminating expenses associated with idle infrastructure.

- **Isolation:** Workloads are executed within sandboxed environments, such as containers or microVMs, ensuring strong isolation between concurrent client requests.
- **Statelessness:** Traditional FaaS functions are ephemeral and stateless. Any persistence required across invocations must be offloaded to external storage services (e.g., object storage or managed databases).
- **Granularity and Lightweight Execution:** Applications are decomposed into small, discrete functions with typically short execution times, ranging from milliseconds to minutes.
- **Event-driven Nature:** Execution is typically triggered by asynchronous events, allowing for highly reactive system designs.

These features have established serverless computing as a robust standard for developing cloud-native applications, particularly those characterized by variable or bursty workloads.

2.2. Towards Edge Computing

Parallel to the maturation of cloud and serverless technologies, the proliferation of Internet of Things (IoT) devices [30] has driven the emergence of *Edge Computing* [34]. While cloud computing concentrates resources in centralized data centers, edge computing decentralizes processing power, placing it closer to the physical location where data is generated.

This paradigm complements traditional cloud infrastructures by addressing limitations inherent to centralized processing. Specifically, edge devices are capable of delivering ultra-low latency responses, a critical requirement for event-driven applications triggered by IoT sensors and actuators in real-time environments.

Consequently, the seamless integration of these paradigms establishes a *Computing Continuum* [18], spanning from centralized Cloud servers down to Edge devices. This continuum allows for the distribution of computation closer to data sources. However, extending the serverless model to the edge presents unique challenges, as edge devices are often heterogeneous and resource-constrained in terms of CPU, memory, and energy compared to their cloud counterparts. Furthermore, traditional heavyweight virtualization techniques are often unsuitable for these constrained environments. Additionally, network deployments in the continuum must be robust, as edge environments are prone to faults and intermittent connectivity, requiring careful consideration to ensure a uniform abstraction

layer for developers.

2.3. Virtualization Paradigms: From Containers to WebAssembly

To ensure isolation between function instances, serverless platforms rely on virtualization to abstract computational resources. Traditional *Hardware-level Virtualization* [28] provides strong isolation by running multiple Virtual Machines (VMs) on a single physical host. However, VMs are typically too heavyweight for the bursty, short-lived workloads of serverless environments due to their significant memory footprint and slow boot times [49].

These limitations led to the dominance of *Operating System-level Virtualization*, or containerization (e.g., Docker [50]). Containers are lighter than VMs as they share the host kernel while isolating the execution environment through namespaces and cgroups. Despite their efficiency in cloud data centers, containers still impose non-trivial overhead, specifically startup latency (cold starts) and image sizes, which become bottlenecks in the resource-constrained, heterogeneous environments of the computing continuum.

To mitigate these overheads, *WebAssembly* (WASM) [25] has emerged as a high-performance alternative offering process-level isolation. Originally designed for the web, WASM provides a sandboxed execution environment that is secure, fast, and architecture-agnostic. In contrast to containers, which package a partial operating system, WASM binaries are compact and rely on a dedicated runtime, analogous to a low-level Java Virtual Machine (JVM) [62], to execute code with near-native performance.

Recent benchmarks demonstrate that WASM is particularly suited for the edge, offering a drastic reduction in image size and memory consumption compared to traditional containerization [32, 51]. By providing a "sandbox" rather than a "container," WASM enables the rapid instantiation of functions on limited devices where traditional virtualization would rapidly exhaust available resources.

2.4. Challenges and Motivating Scenarios

Despite the respective strengths of the cloud and edge paradigms, significant challenges remain when attempting to unify them into a cohesive system. This section analyzes these hurdles through the lens of specific use cases.

2.4.1. The Problem of State Management and Mobility

In the context of this work, we define *state* as the persistent, mutable data associated with a specific application entity or workflow (e.g., user sessions, IoT device status, or ML model parameters) that must be preserved across asynchronous invocations and potentially migrated across the computing continuum. The original serverless programming model is inherently stateless, where ephemeral functions are instantiated solely in response to events. Consequently, there is no native mechanism for state maintenance across function invocations. This restriction severely limits the scope of deployable applications; for instance, systems requiring the tracking of user operation history or scientific workflows that frequently retrieve and modify datasets cannot be easily implemented.

Although functions can be engineered to access external storage explicitly, this approach introduces significant overhead. Developers are forced to manually manage database connections and query logic. Moreover, managing shared state in this manner increases complexity, as synchronization properties must be explicitly handled within the function logic, effectively negating the abstraction benefits of the serverless model.

These limitations have motivated research into stateful serverless systems [5]. In such architectures, the platform provides an internal data storage mechanism exposed through a high-level abstraction layer, allowing functions to interact transparently with state without managing low-level connectivity.

We illustrate this limitation by introducing two scenarios, particularly where data possesses high 'gravity' or must follow the movement of physical entities.

Regional Healthcare Analytics

We consider distributed medical simulations based on complex mathematical models used to monitor disease progression in patient groups. In this scenario, the architecture is strictly hierarchical: cloud data centers provide global coordination, hospitals act as fog nodes for regional data, and specific wards operate as edge nodes for real-time patient data acquisition.

These distributed medical simulations resemble classical high-performance computing (HPC) workloads in terms of computational complexity, which typically employ MPI [36] for tightly coupled simulations within a single data center, but healthcare analytics in a continuum requires a distinct approach. Data privacy regulations and "data gravity" prevent the transmission of raw patient records to the cloud. To address this, a

continuum-aware abstraction is required that allows lightweight simulation functions to execute locally on hospital edge nodes.

By maintaining state, such as a patient's physiological trajectory, locally, we avoid the latency and bandwidth overhead associated with transmitting large datasets. Crucially, only aggregated results or anonymized updates are migrated to the cloud. This approach decouples the local simulation logic from complex global topologies, addressing the unreliability of hospital networks while ensuring compliance with strict data residency requirements.

Smart Logistics

This scenario involves the orchestration of order distribution across a national territory. While orders may originate in the cloud, their fulfillment requires decentralized management via local distribution centers. As a physical package moves through the supply chain, a vehicle, acting as a mobile client, assumes responsibility for it.

The "digital twin" of a package, encompassing information such as delivery status and routing history, must dynamically follow the physical item as it transitions between network nodes. Unlike traditional FaaS solutions, which require developers to manually handle data handoffs between regional databases, a **seamless migration mechanism** would automate state relocation.

2.4.2. The Problem of Cloud-Only Applications

Cloud-only architectures often fail to account for the proximity of execution relative to the user and data sources [26]. From an external perspective, a cloud region functions as a massive, centralized resource pool designed to serve multi-tenant workloads. While latencies within a single data center are negligible, the latency penalty becomes significant when a function must access data located in a geographically distant region or on an edge device.

To address this, it is essential to develop systems capable of dynamically scheduling function execution based on data locality. By enabling data and functions to move transparently between the edge and the cloud, within the Computing Continuum, we can significantly reduce the overall execution latency perceived by the user.

However, migrating computation to the edge introduces reliability concerns. Unlike the stable environment of a cloud data center, edge devices are prone to failures and network instability. Therefore, introducing replication mechanisms distributed along the contin-

uum becomes crucial to ensure data availability and fault tolerance in this heterogeneous environment.

The inefficiencies of cloud-centric architectures become critical in applications requiring either massive data throughput or real-time responsiveness. The following cases of distributed training and urban monitoring highlight the necessity of data-local execution.

Federated Learning

This use case focuses on training large-scale machine learning models using distributed data sources, such as animal species recognition via cameras in remote habitats. The global model is distributed to edge nodes and trained locally, after which updates are aggregated centrally.

In traditional cloud architectures, a function is required to fetch the entire model state from an external storage service (e.g., Amazon S3) at the start of every training batch, only to write the updated state back upon completion. For large models, this I/O overhead is prohibitive. To mitigate this, the underlying platform must allow model weights, representing the application state, to be stored directly in edge components for fast retrieval in subsequent training runs. This approach would significantly reduce overall bandwidth consumption and facilitate the distributed training of complex models by directly leveraging the storage abstraction located on edge devices.

Smart City

In a Smart City environment, thousands of heterogeneous IoT sensors continuously monitor urban infrastructure, including traffic systems, air quality indices, and public transport networks. These devices transmit data to the nearest fog aggregation point for real-time processing.

The primary challenge here is "High-Fan-In" aggregation. In a cloud-centric model, transmitting every sensor reading to a central region creates a "hairpinning" effect, where data travels to the core for processing only to trigger actions back at the edge, wasting bandwidth and increasing latency. A stateful platform capable of local aggregation at the fog level would facilitate faster data processing and response times. This capability is essential to satisfy strict timing requirements in safety-critical environments where decisions or corrective measures must be adopted rapidly. For instance, an edge function can maintain a running average or an anomaly detection window within its managed state, triggering a cloud-level alert only when a specific threshold is met.

2.5. Synthesis of Requirements and Research Objectives

Our analysis of the preceding scenarios reveals a systemic gap in the current computing continuum. While Serverless Computing provides the desired abstraction and Edge Computing provides the necessary proximity, their unification is hindered by two fundamental architectural deficiencies:

1. **The State-Abstinence Gap:** Current FaaS models treat state as an external burden rather than a primary citizen. This forces manual, high-latency data management that is incompatible with the needs of mobile digital twins (Smart Logistics) and data-resident simulations (Healthcare).
2. **The Locality-Awareness Gap:** Cloud-native orchestrators prioritize resource volume over resource proximity. This leads to the "hairpinning" and I/O overheads that make real-time urban monitoring (Smart City) and large-scale distributed training (Federated Learning) inefficient.

To bridge the gap between local data generation and distributed state management, our proposed platform Ermes is designed to fulfill the following research objectives:

- **Integrated State Management:** Transition from external database dependencies to an internal state abstraction, enabling low-latency access and transparent synchronization.
- **Locality-Aware Scheduling:** Implement a continuum-aware scheduler that places computation physically close to data sources to eliminate "hairpinning."
- **Transparent State Migration:** Provide automated mechanisms to relocate digital twins/states alongside mobile clients (e.g., vehicles or patients).
- **High-Responsiveness Replication:** Employ distributed replication strategies across volatile edge nodes to reduce the overall latency perceived by the client.
- **Lightweight Isolation:** Leverage WASM to provide a safe, high-performance execution environment suitable for the resource-constrained nodes identified in our scenarios.

3 | Related Work

The diversification of serverless computing, spanning from centralized data centers to edge-native environments, has resulted in a fragmented landscape of solutions, ranging from mature industrial cloud platforms to specialized academic prototypes designed for the network edge. As the Function-as-a-Service (FaaS) model moves beyond centralized data centers, new challenges regarding state management, decentralized orchestration, and resource efficiency have emerged.

The goal of this chapter is to provide a systematic review of the state of the art to identify the architectural gaps that the solution proposed in this thesis aims to fill. We categorize the existing literature into three main domains: established cloud-based offerings, lightweight execution runtimes, and emerging research frameworks. To facilitate a rigorous comparison across these diverse systems, we first introduce a multi-dimensional classification framework. This framework serves as the basis for the summary provided in Table 3.1 and Table 3.2, which highlight the operational and functional trade-offs of current technologies.

3.1. Classification Framework

Numerous serverless solutions have been proposed in both industry and academia. To provide a systematic comparison, we structure our analysis around a set of architectural, operational, and functional criteria. This framework allows us to evaluate how current technologies address different facets of the Function-as-a-Service model.

The objective of this classification is twofold. First, it serves to clarify the current landscape of available solutions. Second, and more critically for this work, it enables us to systematically identify the weaknesses and missing features in these systems, thereby contextualizing the specific challenges our architecture is designed to address. Specifically, we examine how each solution approaches core aspects such as its execution model, state management, scheduling policies, and resource management techniques. This analysis delineates to map the strengths and limitations that shape both established industrial

platforms and emerging academic prototypes.

By adopting this structured approach, we build a comprehensive understanding of the field. We can identify areas where existing efforts converge on common solutions, where they diverge into competing design philosophies, and, most importantly, where they systematically leave unaddressed the capabilities that our own architecture is designed to provide.

3.1.1. Architectural and Functional Dimensions

Table 3.1 presents an initial classification based on the primary layers of the serverless stack. We categorize *Runtime* solutions as those introducing lightweight virtualization techniques, such as *WebAssembly* (WASM) [25] or microVMs, to enable efficient function execution in resource-constrained edge environments where traditional containerization overhead is often prohibitive. In contrast, *Platforms* represent comprehensive FaaS implementations, which we further distinguish by their deployment scope (cloud-only or edge-supported). Our classification also includes *Extensions*, modular components designed to augment the capabilities of existing platforms, and *Theoretical Models* that provide abstract formalizations of serverless environments.

Furthermore, we identify whether a solution is *Decentralized*, meaning it supports a distributed architecture for deploying serverless functions across diverse network nodes. We mark most major cloud platforms as not providing support ($\times$) for this feature; while these platforms operate across multiple geographic regions to ensure global coverage, orchestrating deployments across these regions typically requires manual intervention or additional configuration.

We also evaluate the capacity of each runtime or platform to support function development in multiple programming languages and analyze its *State Model*. Referring to our definition of *state* presented in Section 2.4.1, we define an architecture as **stateful** if it provides intrinsic mechanisms to manage state preservation across function invocations. Within this category, we examine *State Abstraction*, which refers to the provision of high-level APIs that abstract the underlying storage interactions from the developer. Conversely, we classify systems as **stateless** if they lack native state management and require functions to interact with external storage services explicitly.

	Type	Multi-language	Decentralized	State Model	State Abstraction
Sledge	Runtime	✓(WASM)	×	Stateless	-
TinyFaaS	Runtime	✓(Docker)	×	Stateless	-
Faasm	Runtime	✓(WASM)	×	Stateful	✓
Faasd	Runtime	✓(Docker)	×	Stateless	-
AWS Lambda	Platform (Cloud)	✓	×	Stateless	-
AWS Lambda @ Edge	Platform (Cloud)	✓	✓	Stateless	-
Google Cloud Run Functions	Platform (Cloud)	✓	×	Stateless	-
Microsoft Azure Functions	Platform (Cloud)	✓	×	Stateless	-
Azure Durable Functions	Platform (Cloud)	✓	×	Stateful	✓
Cloudflare Workers	Platform (Cloud)	✓(WASM)	✓	Stateful	✓
AWS IoT Greengrass	Platform (Cloud)	✓	✓	Stateless	-
OpenWhisk	Platform	✓	×	Stateless	-
Lean OpenWhisk	Platform	✓	×	Stateless	-
OpenFaaS	Platform	✓	×	Stateless	-
Crucial	Extension	-	-	Stateful	✓
Cloudburst	Platform	×	×	Stateful	✓
Apache Flink Statefun	Platform	✓	×	Stateful	✓
FaDO	Platform	✓	✓	Stateful	×
EDGELESS	Platform	✓(WASM)	✓(Hierarchical)	Stateless	-
StructMesh	Platform	✓(Docker)	✓	Stateful	×
DFaas	Platform	✓(WASM)	✓	Stateless	-
Funless	Platform	✓(WASM)	×	Stateless	-
Nardelli et al.	Theoretical model	-	✓	Stateful	-
Serverledge	Platform	✓(Docker)	✓(Hierarchical)	Stateless	-
Enoki	Platform	✓(Docker)	✓	Stateful	✓
Vahabi et al.	Platform	✓	✓(Hierarchical)	Stateless	-
Cicconetti et al.	Platform	✓	✓	Stateless	-
Neptune	Platform	✓(Docker)	✓	Stateless	-

Table 3.1: State model overview.

3.1.2. Operational and Management Dimensions

In Table 3.2, we examine the operational dynamics of these architectures. The *Scheduling Policy* determines which component decides where to execute a function. In a *Trivial* policy, the node receiving the invocation handles the execution. A *Centralized* policy relies on a single entity, such as a master node, to manage task assignment. *Distributed* policies allow nodes to cooperate autonomously in decision-making, while a *Hybrid* policy restricts decisions to a subset of nodes. We also determine if the scheduling logic is *Locality-Aware*, accounting for the proximity between the execution node and the user.

Our analysis includes *Resource Management* mechanisms, such as load balancing and dynamic offloading to mitigate congestion. Finally, we indicate whether an architecture supports *Data Replication*, which is a requirement for high availability in distributed environments.

	Scheduling Policy	Locality Awareness	Resource Management	Data Replication
Sledge	-	-	-	-
TinyFaaS	-	-	-	-
Faasm	-	-	-	✓
Faasd	-	-	-	-
AWS Lambda	Centralized	×	✓	-
AWS Lambda @ Edge	Trivial (CDN)	✓	×	-
Google Cloud Run Functions	Centralized	×	✓	-
Microsoft Azure Functions	Centralized	×	✓	-
Azure Durable Functions	Centralized	-	-	∼
Cloudflare Workers	Trivial (CDN)	✓	×	×
AWS IoT Greengrass	Centralized	✓	×	∼
OpenWhisk	Centralized	×	✓	-
Lean OpenWhisk	Centralized	×	✓	-
OpenFaaS	Centralized	×	✓	-
Crucial	-	-	-	✓
Cloudburst	Centralized	×	✓	✓
Apache Flink Statefun	Centralized	×	✓	✓
FaDO	Centralized	✓	✓	✓
EDGELESS	Hybrid	×	✓	-
StructMesh	Centralized	×	✓	×
DFaas	Distributed	×	✓	-
Funless	Centralized	×	✓	-
Nardelli et al.	Distributed	✓	✓	×
Serverledge	Distributed	✓	✓	-
Enoki	Trivial	✓	×	✓
Vahabi et al.	Hybrid	✓	✓	-
Cicconetti et al.	Centralized	✓	✓	-
Neptune	Centralized	✓	✓	-

Table 3.2: Scheduling policy overview.

Following this classification framework, we structure our review as follows. We first discuss lightweight runtime environments designed for efficient function execution at the edge. We then examine the major industrial cloud serverless platforms, which represent the current industry standard. Finally, we survey academic and research-oriented platforms to highlight emerging architectural trends.

3.2. Cloud Solutions

FaaS cloud solutions are normally offered by major cloud service providers through a pay-as-you-go method. Most notable examples are AWS (Amazon Web Services) and their *Lambda Functions* [2], Google with their *Cloud Run Functions* [22] and Microsoft *Azure Functions* [38] None of them intrinsically offer abstractions for state management, and often rely on external services forcing developers to explicitly write the interactions with the databases in the functions.

A variation of Microsoft Azure Functions, denoted *Durable Functions* [37], which provides an abstraction layer for long-running, stateful orchestrations and persistent state management thanks to the open source project *Durable Task Framework* [39]. Nevertheless, the functions are always meant to run on the Microsoft Azure cloud and cannot be deployed on specific edge devices.

Meanwhile, AWS has extended its infrastructure to the network edge by introducing *Lambda at Edge* [3] and *IoT Greengrass* [1]. In particular, the first framework is an extension of their cloud infrastructure, which executes functions on *CloudFront* nodes, strategically located closer to users in order to reduce the communication latency. And IoT Greengrass is an open source framework which can be deployed on custom IoT devices and enables communication and deployment of Lambda Functions with AWS servers, effectively creating a continuum. Nevertheless, there is no abstraction for explicit state management and the framework only works using AWS technologies, prohibiting the integration with other cloud solutions.

Finally Cloudflare *Workers* [14] introduce *Durable objects* [60], which contain a state saved to persistent storage. Requests directed to a specific Durable Object are then forwarded towards the object itself enabling request coordination and state consistency across concurrent invocations. Moreover, these objects can be transparently migrated if needed. Unfortunately, this technology is coupled to the Cloudflare environment and does not allow for the participation of edge nodes.

3.3. Research Work

This section provides an overview of existing research addressing serverless computing in edge environments, categorized by their primary focus: execution models (specifically WebAssembly versus containerization), state management, architectural approaches to decentralization and resource optimization, and specific WebAssembly runtimes tailored for these demanding environments.

3.3.1. WebAssembly Runtimes for Edge Environments

To enable isolation for function executions in resource-constrained environments where conventional VMs and containers are unsuitable, WebAssembly (WASM) has emerged as a promising technology. We discuss prominent runtime options for WASM in this context.

Sledge [20] introduces its own LLVM-based compiler for WASM and provides a runtime where applications execute within a lightweight sandbox. Functions are dynamically loaded upon invocation, with the runtime managing safe memory allocation. To minimize memory footprint, the runtime and libraries are shared across all functions, and startup time is reduced by decoupling the linking and loading process from function instantiation. Function inputs and outputs are formatted as HTTP requests.

Faasm [57] introduces *Faaslets*, which are lightweight isolation abstractions allowing functions to run in the same address space while maintaining isolation through WASM compilation for efficient state access. Additionally, it supports a "read global - write local" filesystem, enabling functions to read from a global object and write to locally cached file versions. Changes in local files are committed to the global file via a push operation, with consistency guaranteed by read/write locks.

3.3.2. Execution Models: WebAssembly vs. Containers

Several platforms leverage containerization for function execution. *OpenWhisk* [4] is a serverless platform that supports both manual and event-triggered function invocations, distributing executions across Docker-wrapped functions. Its lean version, *Lean OpenWhisk* [8], optimizes for resource-constrained environments by reducing its footprint and consolidating components. Similarly, *OpenFaaS* [43] employs a robust internal system to manage Docker-containerized functions. Its daemon, `faasd`[1], enables deployment without *Kubernetes*, making it suitable for edge nodes. *TinyFaaS* [45] is a lightweight platform designed for stateless serverless functions on edge devices. It handles client connections and load balances them to Docker-containerized function handlers, ensuring application-level isolation. *Serverledge* [53] addresses FaaS distribution in the Edge-to-Cloud continuum by partitioning the space into cloud regions and edge zones. Each node can execute functions locally or offload them, employing both horizontal and vertical offloading policies for containerized functions, which remain stateless.

In contrast to container-based approaches, *Funless* [16] employs WebAssembly (WASM) to reduce virtualization overhead at the edge. Its architecture is divided into components

[1] https://github.com/openfaas/docs/blob/master/docs/deployment.md

responsible for request reception, function binary storage, and function deployment. However, functions within this platform also remain stateless.

3.3.3. State Management in Serverless Functions

Recent research has focused on state management as a primary bottleneck for edge-based serverless performance. *Crucial* [6] proposes a distributed shared objects abstraction, providing a common global state accessible by functions deployed on cloud nodes, guaranteeing strong consistency (wait-free and linearizable) using a DHT for storage. While not a platform itself, it offers a state abstraction layer for existing cloud serverless platforms. *FaDO* [58] is an orchestrator for stateful serverless functions, where data is organized into buckets and replicated across different computing clusters. It offers customizable load balancing policies to match functions with data buckets, albeit with a single entry point for client requests. *EDGELESS* [13] focuses on stateful functions at the edge by passing state between consecutive function invocations within a workflow. It features a hierarchical structure with computing domains, where clients connect to initiate workflows. *Enoki* [46] introduces stateful functions to the edge by using *FReD* [47] key-value storage and *TinyFaas* as the FaaS platform.. It guarantees client-side consistency through local data access and replication.

3.3.4. Decentralization and Resource Optimization

Decentralization and optimized function placement in edge environments are explored by other works. *DFaas* [11] decentralizes execution among edge nodes using a Peer-To-Peer network that dynamically offloads functions based on node workload. The network is organized into federated regions, each hosting a platform portion managed by agents that handle client requests and load balance them. This approach currently uses a linear load balancing policy and does not address persistent data. *StructMesh* [9] models a pipeline for distributing computation and data across the computing continuum, from edge to cloud nodes, enabling parallel processing and load balancing through its mesh structure. *Nardelli et al.* [40] present a novel function scheduling policy and formalize data placement optimization with respect to client requests from different nodes. *Vahabi et al.* [59] propose a model for edge function execution that minimizes energy consumption through a dynamic scaling approach. *Cicconetti et al.* [12] introduce a model for function execution and offloading in a full edge environment, featuring a dynamic edge layer constantly updated by dedicated nodes providing topology and network information. Their work also explores dynamic function scheduling policies and demonstrates how a hierarchical network can improve scalability by reducing routing table dimensions. *Nep-*

tune [7] addresses stateless serverless function placement in edge environments with high client mobility and latency constraints. Its network is structured into various hierarchical levels that exchange information through a common controller. The main idea is that nodes dynamically decide resource allocation, and a higher-level entity is responsible for dynamically routing the workload.

3.4. Beyond the State of the Art: Conclusions and Motivations

The analysis of existing solutions highlights substantial progress in serverless computing, spanning from lightweight WASM runtimes to enterprise-grade platforms and innovative research prototypes. Nevertheless, a recurring pattern emerges: while many solutions excel in isolated aspects, such as low-overhead execution (WASM-based approaches), orchestration and scalability (cloud platforms), or distributed scheduling and function placement (edge-oriented research), none provide an integrated architecture capable of jointly supporting stateful execution, decentralized coordination, edge-cloud continuum deployment, and fine-grained resource optimization. Current cloud platforms offer strong orchestration capabilities but rely heavily on external storage and centralized control, limiting adaptability in distributed and heterogeneous environments. Conversely, academic works address decentralization and edge deployment but often rely on stateless functions, or provide state abstractions decoupled from computation, leading to suboptimal performance for data-intensive or strongly consistent workloads.

This gap highlights the need for a unified solution that treats state as a first-class architectural concern, rather than an external dependency. Such a system must supports transparent state locality and migration, and enables dynamic function scheduling based not only on computational load but also on data proximity and consistency requirements. Moreover, despite the emergence of WASM runtimes, most systems fail to leverage WASM's potential for stateful, secure, and efficient edge-native execution. This gap defines the motivation of our work: we propose an architecture that seamlessly bridges function execution and state management across heterogeneous environments, enabling distributed decision-making, adaptive placement, and true integration between computation and data. In doing so, our approach attempts to overcome the limitations observed in existing systems, potentially providing a foundation for a new class of state-aware serverless platforms explicitly designed for the computing continuum.

4 | Ermes Framework

The Ermes framework is a modular architecture designed to support stateful computations across a distributed infrastructure. This chapter provides a comprehensive overview of this architecture, deconstructing it into three foundational pillars.
We begin with the **Architectural Model and Infrastructure Topology**, which establishes the static topology of nodes and the roles of our core system services. This static infrastructure is crucial for our stateful design, as it provides a stable foundation for predictable lookup and scheduling.
We then present the **Data Model and Consistency Guarantees**, which defines the fundamental abstractions for state persistence, organization, and consistency. Finally, the **Execution Model and Lookup** synthesizes these elements to describe the system's dynamic behavior, detailing the complete lifecycle of a serverless function invocation. This chapter lays the conceptual groundwork for the implementation and evaluation presented in the remainder of this thesis.

4.1. Ermes Overview and Design Principles

The Ermes framework facilitates the deployment of stateful serverless functions, compiled from diverse source languages into an generic intermediate representation, across a distributed infrastructure. Within the proposed FaaS model, we distinguish three primary roles: *clients*, who trigger function execution; *developers*, who define function logic and manage associated data via the Ermes API; and *network providers*, who design the physical and logical topology to ensure geographical coverage and enforce execution constraints.
Our primary objective is to minimize user-perceived latency during function invocation. To this end, we employ a data-centric scheduling model: by associating functions with specific data subsets, the system schedules execution on nodes that host at least a portion of the required state.
Simultaneously, Ermes aims to streamline the development lifecycle by abstracting the complexities of authentication, artifact retrieval, placement, and execution. We realize these system goals through a strict architectural separation between the management

layer, which governs user-data associations, and the distributed execution layer, as detailed in the subsequent sections. This separation facilitates centralized administrative oversight while supporting efficient, on-demand execution across the network. Furthermore, by decoupling function deployment from execution, we enhance system flexibility and simplify update management, thereby improving the infrastructure's elasticity and its ability to adapt to dynamic workloads.

To achieve low-latency execution across a geographically distributed and shared infrastructure, the system must balance performance, availability, and correctness across heterogeneous workloads. We expose configurable data replication and consistency models, allowing developers to explicitly control the trade-off between availability and correctness according to specific application semantics. In this context, data access control becomes a first-class concern. Since functions execute in proximity to data and may span multiple administrative domains, the system must enforce precise visibility and sharing guarantees on the application state. To address this, Ermes provides three distinct data access control policies:

- **Private**: Application data is accessible exclusively by its owner, ensuring strong isolation for sensitive or user-specific state.
- **Shared**: Application data is accessible by a predefined set of user groups, enabling controlled collaboration while preserving access boundaries.
- **Public**: Application data is accessible by any user without restriction, supporting open and read-mostly workloads.

As detailed in Section 4.2.2, we enforce these policies through our Role-Based Access Control (RBAC) mechanism, which relies on *Group Tokens* issued by the GTP.

These design principles establish the conceptual foundation of Ermes. In the following section, we describe how they are concretely realized through a modular architectural model, detailing the system components and interactions that enable stateful, low-latency serverless execution.

4.2. Architectural Model

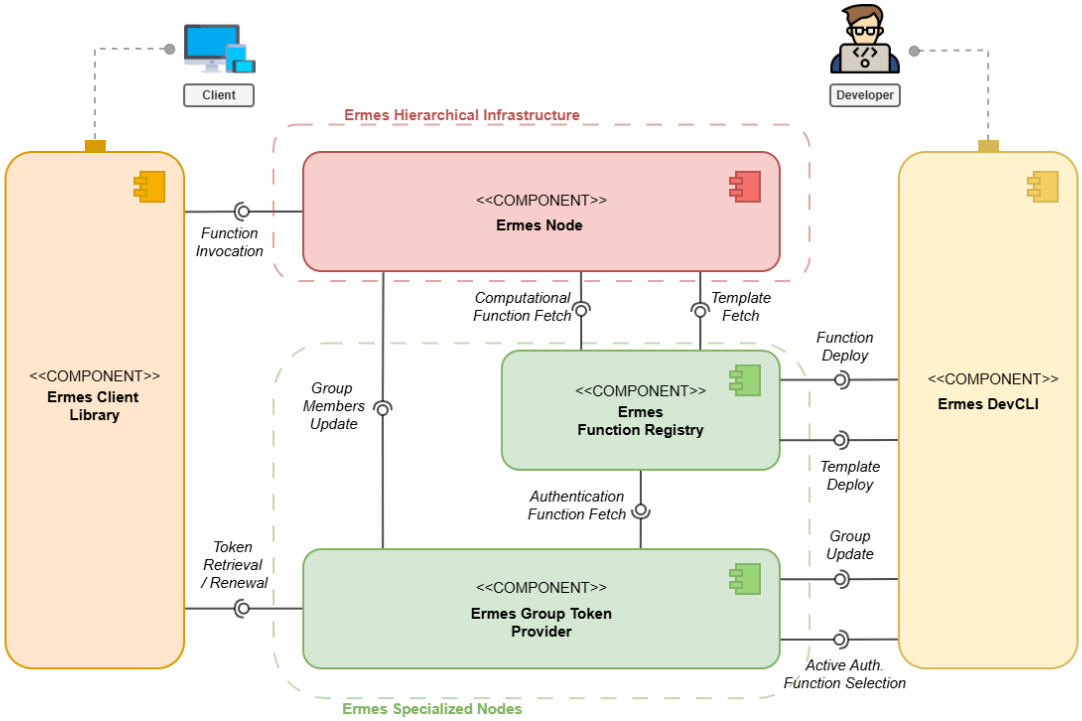


Figure 4.1: Ermes Framework Architecture.

We designed the Ermes architecture as a modular system composed of several interconnected software components. Each component has a well-defined role and exposes clear interfaces for interacting with others. This separation of concerns enhances maintainability and supports scalability. Figure 7.3 provides a high-level overview of these components and their relationships.

The primary entry points to our infrastructure are two external tools: the *Ermes Client Library* and the *Ermes Developer Command Line Interface (DevCLI)*. The Client Library abstracts the complexity of the underlying architecture from client applications; its responsibilities include managing authentication and authorization tokens and implementing client-side caching to support session guarantees. The DevCLI is a developer-focused tool that provides a unified interface for managing the entire serverless function lifecycle.

These tools interact with a distributed network of *Ermes Nodes*, which form the core of the serverless execution layer by running functions and managing local state. Supporting this distributed layer are two centralized components: the *Group Token Provider* (GTP), which handles access control, and the *Ermes Function Registry* (EFR), which manages function storage and versioning. Together, these components ensure consistent

coordination across the system.

The following sections explore each of these core architectural components in detail, describing their functionality, interactions, and role within the overall system.

4.2.1. Ermes Node

In our architecture, we define a *Node* as an independent software process, a design choice that permits a single physical or virtual machine to host multiple concurrent node instances. Each node exposes a public interface for client requests, such as function invocations, and a private interface for internal communication with other nodes in the system. We designate as *Ermes Nodes* those entities whose primary responsibilities are to execute serverless functions invoked by users and to provide storage capabilities for stateful computations.

We designed these nodes with a clear separation between the execution and state management layers. This architecture allows a node to process multiple concurrent requests efficiently while ensuring the integrity of both local and replicated data. *Ermes Nodes* also support the dynamic reallocation of data collections, enabling the system to migrate data across the network to balance load and move it closer to clients. This capability is crucial for maintaining high availability and performance in heterogeneous and evolving environments.

4.2.2. Group Token Provider

The *Ermes Group Token Provider* (GTP) is a core component of our infrastructure that provides centralized persistent storage for group-user associations. The Role-Based Access Control (RBAC)[54] mechanism uses these associations to issue or renew cryptographically signed tokens, which we designate as *Group Tokens*.

Upon request or renewal, the GTP issues *Group Tokens* to users who have been authenticated via external services. Each token encodes data access rights by listing the groups to which the user belongs. To ensure authenticity and integrity, the GTP digitally signs every token, a property that also enables them to be safely cached by clients.
Additionally, tokens have an expiration time, after which a user must renew the token to maintain data access. The expiration time is configurable, allowing Ermes to adapt to different applications and scopes. This configurability, however, introduces a significant trade-off. A shorter expiration time enables faster propagation of group membership changes, and therefore quicker access right revocation, but at the cost of requiring more

frequent token renewals. This higher renewal frequency increases the load on the GTP and can cause it to become a performance bottleneck.

When a function is invoked, the request must be accompanied by a valid *Group Token*. The receiving *Ermes Node* then verifies the token to check the user's data access rights before executing the function. This process enforces RBAC policies in a decentralized yet consistent manner.

This architectural approach offers several distinct advantages. By centralizing the management of group-user associations while distributing enforcement responsibilities to individual nodes, the system streamlines permission management in large-scale deployments and significantly reduces administrative overhead. Group membership updates propagate automatically through the token renewal process, facilitating flexible and dynamic access policies without requiring modifications to individual user permissions. Moreover, the reliance on cryptographically signed tokens enables nodes to verify access rights independently, supporting decentralized enforcement while preserving global consistency and security. Ultimately, the GTP enables Ermes to scale securely, adapt to evolving application requirements, and maintain fine-grained control over data access within a distributed environment.

4.2.3. Function Registry

Another core component of the Ermes infrastructure is the *Ermes Function Registry* (EFR). It plays a key role by storing serverless functions binaries, along with their corresponding *Function Metadata* and *Query Templates* (see Sections 4.4.2 and 4.4.3). For instance, *Function Metadata* includes version numbers to support function versioning. This feature allows multiple versions of the same function to coexist, which facilitates gradual rollouts and rollback strategies.

The EFR provides functions, metadata, and query templates to Ermes Nodes on demand. Developers deploy new or updated functions by uploading their binaries and the associated metadata and templates to the EFR. This process registers the function within the infrastructure, making it immediately available for execution. To minimize latency, nodes cache these artifacts locally after fetching them from the registry for the first time.

Collectively, the GTP and the EFR constitute the management layer, effectively decoupling system coordination from distributed execution. While these centralized components govern access control, artifact storage, and versioning, the distributed network of Ermes Nodes acts as the execution layer, handling function invocation and state management. This architectural separation ensures centralized administration without compromising

scalability, enabling independent deployment and update cycles for functions across the infrastructure.

4.3. Infrastructure Topology

To decentralize computational execution, the framework organizes *Ermes Nodes* into a multi-tiered hierarchy spanning the edge-cloud continuum. Within this structure, each node is assigned a single parent and may manage multiple children. Consistent with established edge computing paradigms, we assume that computational and storage capacities increase as we ascend the hierarchy from the edge toward the cloud [27].

It is important to note that the centralized components, specifically the *Group Token Provider* (GTP) and the *Ermes Function Registry* (EFR), reside outside this hierarchical topology. Operating as architectural singletons, they function independently of the tree structure and offer flexible deployment options at any network location, although they are typically hosted within the cloud layer for high availability.

While our design generalizes to hierarchical structures of arbitrary depth, in this work we focus on a conventional three-tier architecture. The tiers are defined as follows:

- **Tier 3 (Cloud):** The highest level of the hierarchy, composed of high-capacity nodes comprising one or more Cloud Region. Each Cloud Region may include multiple data centers interconnected via high-bandwidth links. While the overall Ermes infrastructure is hierarchical, this layer operates as a dense mesh topology, where nodes communicate directly, enabling efficient data sharing and coordination across cloud resources.
- **Tier 2 (Fog):** An intermediate tier containing nodes that function as aggregation points for the edge. We designed our infrastructure following the assumption that typically fog nodes have less resources than cloud and more than edge.
- **Tier 1 (Edge):** The layer closest to end-users. Edge nodes serve as the primary entry points for client requests and can execute functions requiring data access.
- **Tier 0 (Client Layer):** While client applications form an essential component of the system's operation, they are external to the Ermes infrastructure. Communication with the platform occurs through the *Ermes Client Library*, which intermediates between client logic and the system's core services.

Clients can connect to any *Ermes Node* in the network to request the execution of serverless functions. Given that client workload is dynamic, with users connecting from various

geographical locations, our system is responsible for dynamically migrating data to minimize user-perceived latency.

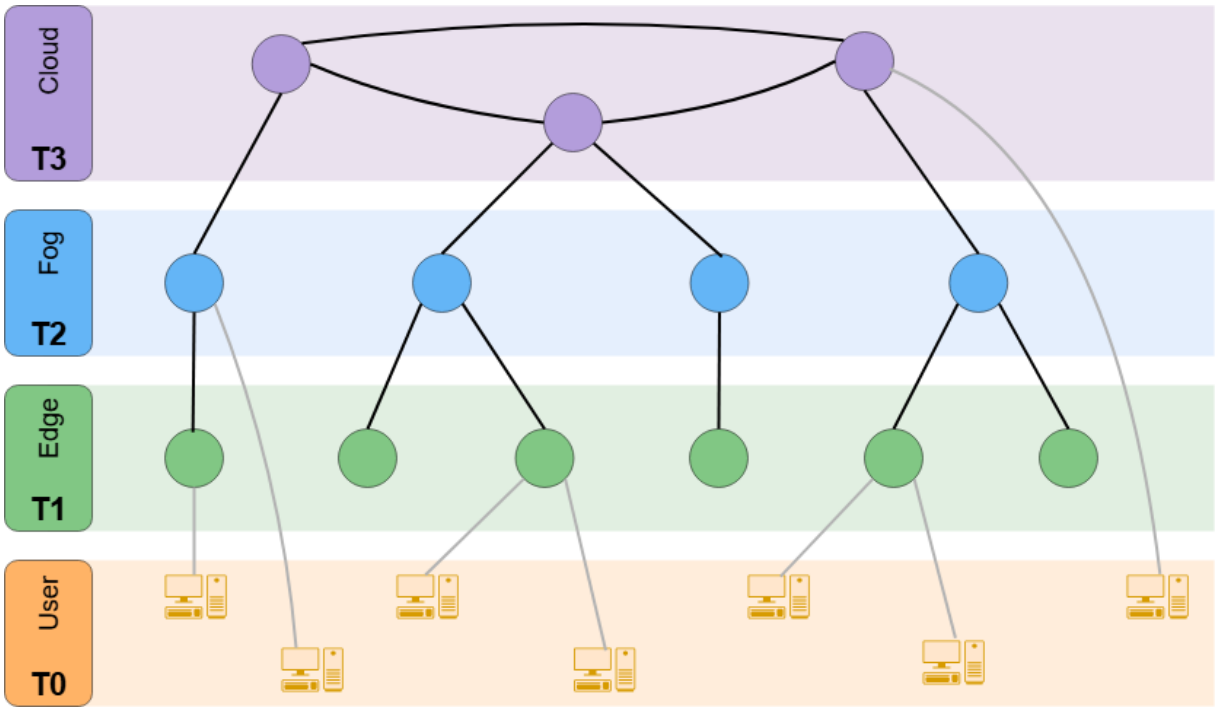


Figure 4.2: Ermes Hierarchical infrastructure.

4.4. Data Model

As Ermes provides a platform for the execution of stateful serverless functions, we define the *state* of a function as the persistent data that endures across multiple invocations and is shared among users within the same group. This state is explicitly managed within a backing storage system, serving as the authoritative source of truth for the application. Crucially, we distinguish this persistent state from a function's *local variables*, which are bound to the lifecycle of a single invocation. While local variables are ephemeral and discarded upon function termination, persistent state remains durable, retrievable across execution sessions, and accessible for concurrent modification. Consequently, developers utilize persistent storage to preserve application-level semantics, while restricting local variables to transient computations within a single execution unit.

The management of persistent state in a distributed environment introduces significant complexity. Specifically, data items belonging to distinct users or applications may share identical identifiers, creating namespace collisions and complicating data isolation. Furthermore, efficiently distributing data across nodes, managing migrations in response to

workload fluctuations, and ensuring high availability necessitate a clearly defined unit of granularity. Without such a structural abstraction, achieving scalability and fine-grained control over data placement and replication becomes increasingly difficult.

To address these challenges, we partition the persistent state space into logical units termed *collections*. A collection aggregates related state elements and serves as the atomic unit for placement, migration, and replication across the system nodes. We uniquely identify each collection using the tuple `<collection_name, owner_id>`, a composite identifier that enables the coexistence of collections with identical names owned by different users.

This architectural design yields several distinct advantages. First, it guarantees semantic isolation: state elements associated with different groups or applications remain logically separated, thereby mitigating conflict risks and simplifying consistency reasoning.
Second, the collection abstraction facilitates flexible partitioning policies. As collections constitute the atomic unit of placement, Ermes can assign them to specific nodes based on workload characteristics, access frequency, or application-specific constraints, supporting fine-grained control over data distribution.
Third, collections enable dynamic scalability, as they can be transparently migrated to balance load across the infrastructure. Finally, by serving as the fundamental unit of replication, collections support high availability through targeted duplication, with replication strategies that can be tailored to specific application requirements.

4.4.1. Consistency Model

Since data collections can be replicated across nodes for availability and performance, and may be accessed concurrently by different users, defining appropriate consistency semantics is essential. To address this, Ermes provides configurable consistency models, allowing developers to balance the trade-off between latency and data correctness.

For each collection, we support two consistency models, whose definitions follow the classification in "Consistency in Non-Transactional Distributed Storage Systems" [61]:

- **Strong Eventual Consistency**: This model, which from now on we refer as *Eventual Consistency*, prioritizes availability and low execution latency by allowing replicas to be updated independently. It guarantees that, in the absence of new writes, all replicas will eventually converge to the same state. However, this approach can lead to write conflicts, which must be addressed with a conflict resolution strategy.
- **Sequential Consistency**: When this model is selected for a collection, our plat-

form prevents write conflicts by enforcing a global order on all write operations across replicas. This guarantee comes at the cost of higher write latency.
To correctly handle client migrations, our Client Library also provides collection-specific *Session Guarantees*, including Read-Your-Writes, Write-Your-Read, Monotonic Reads, and Monotonic Writes [61].

Regarding the implementation of sequential consistency in distributed environments, two primary approaches are typically considered [30]: quorum-based (leaderless) and single-leader replication. Consensus protocols in leaderless systems delegate the task of replicating updates to users; read and write operations are considered complete only after contacting multiple replicas to resolve potential conflicts. This mechanism generally introduces higher operation latency. Conversely, under the single-leader model, all write requests are directed to a designated leader replica, which asynchronously propagates updates to follower replicas. Read-only functions, in contrast, can be served by any available replica.

Furthermore, while our system supports configurable consistency per collection, it does not provide a transition model between them. Therefore, maintaining coherent operations across multiple collections remains the responsibility of the application logic, which must explicitly handle ordering, atomicity, and potential consistency anomalies when performing multi-collection transactions or cross-referenced updates.

4.4.2. Metadata

The management of distributed functions and collections within the Ermes architecture presents specific challenges. First, the system must efficiently associate functions with their execution artifacts, such as binaries and query templates, and resolve dependencies between functions and data collections. Second, to satisfy placement policies and availability requirements, data collections may be dynamically migrated or replicated across nodes, complicating resource discovery and the consistent enforcement of access rules. Without a structured metadata management approach, lookup operations would become inefficient, potentially compromising security and consistency guarantees.

To address these complexities, we employ two distinct categories of metadata in Ermes: *Function Metadata* and *Collection Metadata*. The former facilitates the efficient resolution of dependencies between functions and data collections, ensuring correct execution and accurate function-to-data mapping. The latter governs data collections, defining access control policies and enabling efficient lookup operations.

We manage and store *Collection Metadata* separately from the actual data records. While

data collections can be moved or replicated across the network to meet application requirements or placement policies, we handle their corresponding metadata differently to optimize lookup performance (see Section 4.5.3) and maintain a clear separation of concerns. Specifically, we anchor a collection's metadata to the node responsible for its designated geolocation, and subsequently replicate that metadata upward through the node hierarchy. Access rules are an exception: although part of the *Collection Metadata*, they are also replicated and moved along with the data itself. This design ensures that the node hosting the data has the necessary information to perform group token verification upon access (see Section 4.5.1).

Each collection is defined by a standardized set of attributes detailing ownership, access control, and consistency guarantees. These fields are itemized in Table A.1 in the Appendix. Furthermore, to support expressive, application-specific discovery, we allow developers to define custom metadata fields during collection creation. Once deployed, these custom fields remain immutable to ensure consistency across the distributed infrastructure.

Similarly, *Function Metadata* is defined by developers at deployment time. This metadata resides within the *Ermes Function Registry* (EFR) alongside the function binaries and *Query Templates*, enabling the scheduler to resolve dependencies between functions and data collections. The complete schema of function metadata fields is detailed in Table A.2 in the Appendix.

4.4.3. Query Model

To locate and retrieve *Collection Metadata*, we define a query model that is conceptually similar to traditional database queries but follows a template-based pattern.
Since the specific data required for an execution may not be known until runtime, our system employs *Query Templates*, which allow a function's data dependencies to be resolved dynamically at invocation time. This is possible because, queries may be specified explicitly by the client within the invocation request.

We associate one or more *Query Templates* with each serverless function, storing them in the EFR as part of the function's metadata. At invocation, the client's request supplies the parameters to complete a template, which generates a concrete *Metadata Query*.

Resolution of *Metadata Query* is executed during the Lookup phase (Section 4.5.2), returning a set of candidate collection identifiers that satisfy the query's conditions. Our system then performs an access control check on these candidates by validating the invoking user's group memberships against the access rules defined in each collection's

metadata. The resulting list of authorized collection identifiers is passed to the Data Discovery phase (Section 4.5.3), which is responsible for retrieving the data required for the function's execution.

4.5. Execution Model

The execution model of Ermes governs the dynamic behavior of serverless functions within our distributed infrastructure, defining the progression of invocations from the initial client request to completion. In contrast to traditional stateless FaaS platforms, Ermes explicitly supports stateful computations across a geographically distributed topology. This architectural choice necessitates robust mechanisms to address challenges related to data locality, consistency models, and coordinated access to shared state.

To address these complexities, we decompose the function lifecycle into five distinct logical phases: authentication and authorization, metadata lookup, scheduling, function execution, update propagation and consolidation. Each phase is engineered to enforce security policies, ensure correctness, and optimize performance across distributed nodes. The *authentication and authorization* phase restricts function invocation and state access to verified entities. *Metadata lookup* efficiently resolves resource dependencies, leveraging caching mechanisms to identify relevant data and minimize latency. *Scheduling* utilizes data locality information and system state to identify the optimal execution node, balancing latency minimization with consistency requirements. Finally, the *execution* phase performs the actual computation, managing both local and remote data access while propagating updates according to the specified consistency semantics.

This section provides a detailed analysis of each phase, delineating the mechanisms that enable low-latency, stateful serverless execution. By modeling the interactions among user requests, system metadata, and data replicas, we bridge the gap between the conceptual design principles and the concrete operations executed by the *Ermes Nodes*, the *Group Token Provider*, and the *Function Registry*. Ultimately, this model demonstrates how the system achieves secure, efficient, and predictable function execution within a distributed environment.

4.5.1. User Authentication and Authorization

In our infrastructure, we delegate user authentication to an external *Identity Provider* (IdP), while authorization is managed internally through a role-based access control model.

The authentication flow begins when a user obtains an *Authentication Token* from the IdP. To validate the user's identity, this token is processed by a developer-provided *authentication function*. Once the identity is confirmed, the authorization phase proceeds. The *Group Token Provider* (GTP) issues a cryptographically signed *Group Token* which explicitly encodes the user's access rights by associating them with one or more groups. Access to data collections and serverless functions is subsequently granted based on this group membership.

The *Ermes Client Library* caches the *Group Token* and attaches it to all subsequent function invocations. When a node receives a request, it decodes the token, extracts the user's group memberships, and verifies that the user is authorized to access the requested collections.

The complete procedure is illustrated in Figure 4.3.

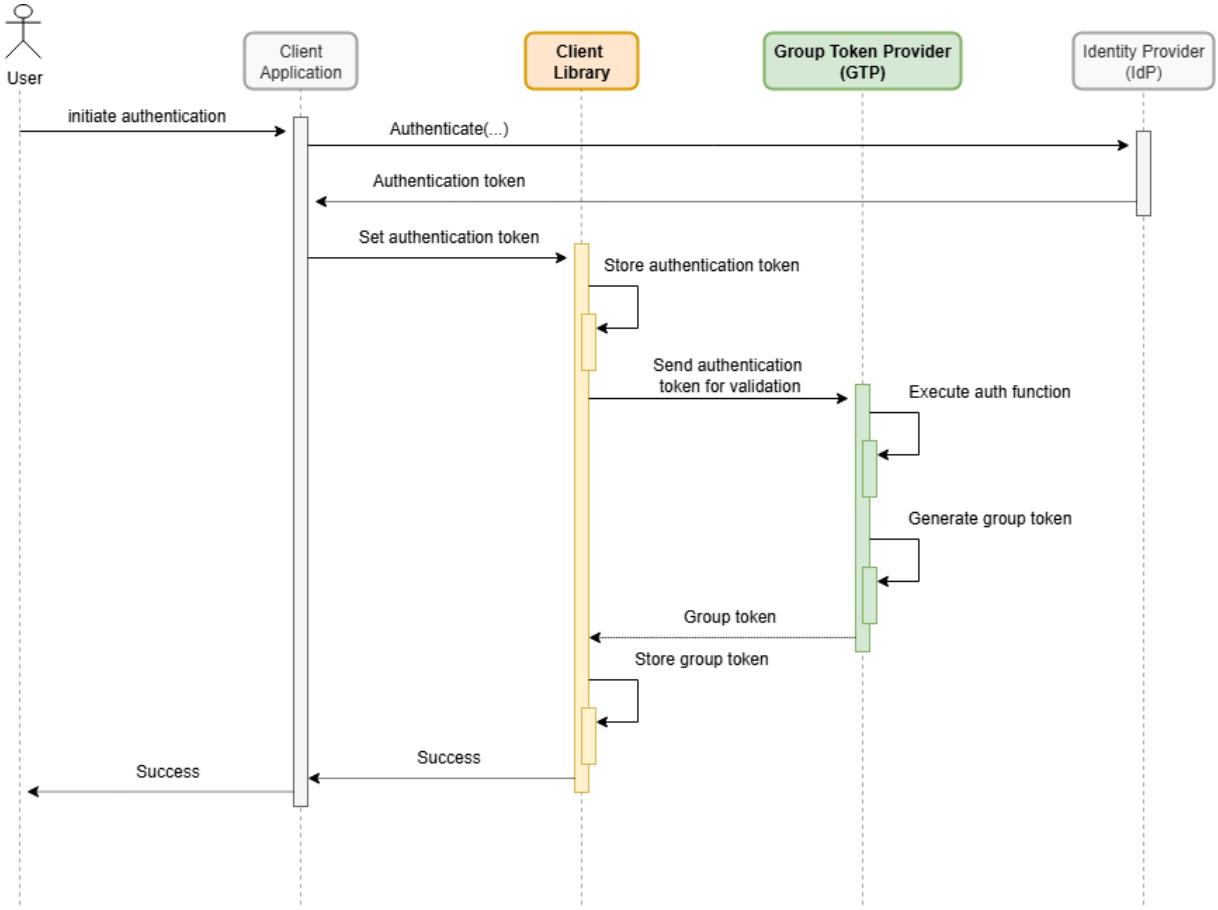


Figure 4.3: Authentication and Authorization procedure.

4.5.2. Metadata Lookup

In this phase, the Ermes framework resolves the collection identifiers required for data retrieval and function execution. Our system queries the node hierarchy to retrieve the metadata of collections that the function will access, ensuring that each function can dynamically identify its data sources based on the invocation context and user permissions before execution begins.

The Metadata Lookup process starts after an *Ermes Node* receives a function invocation request and validates the accompanying *Group Token*. From the request, the node extracts a fixed set of parameters that guide the lookup procedure, including authorization information, optional locality hints, and the identifiers and parameters of the metadata query templates. These parameters are summarized in Table A.3 in Appendix.

Once these parameters are obtained, the receiving node determines the query's execution path. If no geolocation is specified, the query is delegated to the cloud location, which maintains a global view of all metadata within their region and can guarantee a successful resolution.
If a geolocation is provided and the current node is responsible for that domain, it executes the query locally. Otherwise, it propagates the request up the hierarchy.
This mechanism enhances scalability and correctness: queries are served locally whenever possible to reduce latency, while hierarchical propagation ensures that any query can be resolved at the cloud level.

After the data dependencies are resolved, the resulting collection identifiers are cached on the client side to bypass the Metadata Lookup phase in subsequent invocations.
This approach reduces latency for repeated function calls but introduces a trade-off, as modifications to collections that involves changes in the data space partition structure (e.g., creations or deletions) are not visible to the client until the cache expires and the lookup is re-executed.
It is important to notice, this caching strategy does not introduce security concerns. Although the cached identifiers may become stale with respect to authorization changes, the user's live *Group Token* is always re-validated before any data is accessed, ensuring that security policies are consistently enforced.

4.5.3. Scheduling Process

The scheduling process is a key component of our framework, responsible for selecting an execution node for each function. As discussed in prior work [33], scheduling in a

distributed environment presents a fundamental trade-off between locality and fairness. Prioritizing locality minimizes invocation latency by co-locating functions with their data, thereby reducing communication overhead. Prioritizing fairness, in contrast, ensures that no single application or user monopolizes system resources. An effective scheduler must balance these competing objectives to optimize overall system efficiency.

Unlike traditional stateless serverless platforms, our infrastructure is stateful and must account for data consistency requirements. Consequently, our scheduling decisions prioritize data locality to reduce network latency by minimizing the number of remote accesses performed on collections. This primary goal is balanced against node availability and the need to respect the consistency semantics (Sequential or Eventual) defined for each collection (Section 4.4.1).

The scheduling process is initiated after the system identifies the required data collections. This occurs in one of two ways: if a Metadata Lookup was performed, scheduling begins on the node that completed the lookup; if the client provided cached collection identifiers, scheduling starts on the node that initially received the invocation request.

Our framework does not provide a general fault tolerance or retry mechanism. However, we make one specific exception to enforce client *Session Guarantees*. If an access to an eventually consistent collection would violate a *Session Guarantee* due to an outdated replica, the node pauses execution. It waits to receive the necessary updates before proceeding with the function execution.

Data Discovery

The first step in our scheduling process is Data Discovery, where the framework identifies a suitable node for function execution, prioritizing data locality. This phase operates under four key assumptions:

- Each node maintains knowledge of the data collections hosted within its children's subtrees.
- Accesses to eventually consistent collections can be performed remotely, whereas accesses to a sequentially consistent collection must be executed locally on a node hosting a replica.
- Sequentially consistent collections employ a single-leader replication mechanism; consequently, functions requiring write operations must directly access the leader.
- We restrict functions to accessing at most one sequentially consistent collection, although they may access any number of eventually consistent collections. This

constraint is introduced to avoid the need for distributed transaction protocols, which would be required to coordinate operations across multiple strongly consistent data partitions. This design simplifies scheduling by eliminating the need for complex co-location or dynamic migration strategies to guarantee local execution.

Based on these assumptions, the Data Discovery strategy depends directly on the consistency requirements of the collections accessed by the function.

If a function accesses only eventually consistent collections, our scheduler can place it on any node that hosts at least one of the required collections, as shown in Algorithm 4.1. When a node receives an invocation request, it first checks if it stores any of the required collections locally (lines 2-4). If so, it executes the function. If not, the node consults its knowledge of the data layout within its children's subtrees and forwards the request to the appropriate child, if known (lines 5-12). If no local or child replica can be identified, the node propagates the request to its parent (lines 13-14).

While accesses to non-local collections are handled remotely, this introduces network latency. Optimal performance is therefore achieved when all required data is co-located with the execution node.

Algorithm 4.1 Data Discovery for Eventually Consistent Collections (Pseudo-code)

```
1: receive function invocation or redirection
2: if node stores a collection locally then
3:     execute function locally;
4:     return
5: else if node knows a collection direction then
6:     if direction = direction_invoker then
7:         respond instructing execution;
8:         return
9:     else
10:        forward request to correct child;
11:        return
12:    end if
13: else
14:    redirect invocation to parent;
15: end if
```

Conversely, if a function requires access to a sequentially consistent collection, that collection's location dictates the placement decision, as detailed in Algorithm 4.2. Before

initiating discovery, our scheduler inspects the function's metadata to determine if it will perform write operations. This distinction is critical because writes must be directed to the collection's leader replica (lines 2-4 & 8-15), whereas reads can be served by any available replica (lines 5-7 & 16-23).

The discovery process begins with the receiving node checking if it hosts the appropriate replica locally (lines 2-7). If the check succeeds, the Data Discovery phase concludes, and the function is executed on the current node. If the required replica is not local, the node attempts to redirect the request. It consults its routing information to forward the request to the appropriate child node if the destination is within its subtree (lines 8-23). If the collection cannot be located within its subtree, the request is propagated to the node's parent as a fallback mechanism (line 24-25).

Algorithm 4.2 Data Discovery for Sequentially Consistent Collections (Pseudo-code)

```
1: receive function invocation or redirection
2: if node is leader ∧ collection is local ∧ want_write then
3:     execute function locally;
4:     return
5: else if collection is local ∧ ¬want_write then
6:     execute function locally;
7:     return
8: else if node knows direction of sequential collection leader ∧ want_write then
9:     if direction = direction_invoker then
10:        respond instructing execution;
11:        return
12:    else
13:        forward request to correct child;
14:        return
15:    end if
16: else if node knows direction of sequential collection read replica ∧ ¬want_write then
17:    if direction = direction_invoker then
18:        respond instructing execution;
19:        return
20:    else
21:        forward request to correct child;
22:        return
23:    end if
24: else
25:    redirect invocation to parent;
26: end if
```

Function Execution

Following the Data Discovery phase, which selects the execution node, the function execution begins. Our framework provides transparent access to all required collections, abstracting their physical location from the function logic.

Based on our scheduling policy, accesses to a sequentially consistent collection are always local to the execution node. For eventually consistent collections, data is accessed locally if the node hosts a replica; otherwise, the system initiates a remote access. To prevent data from being moved during execution, our system locks any locally accessed collection

for the duration of the function's execution.

When handling remote access requests, nodes act as transparent proxies. If a node receives a request for a collection it hosts locally, it serves the request directly. If not, it consults its hierarchical view and forwards the request to the next hop in the hierarchy (either a parent or a child). This hop-by-hop forwarding ensures that data can be located even if the requester has an incomplete or outdated view of the collection's location.

To reduce the load on the underlying datastore and minimize the frequency of remote requests, we provide a function-private, write-back cache. All write operations performed by a function are initially buffered in this local cache. They are persisted to the key-value store and propagated to other replicas only in a final commit phase after the function has completed its execution.

For sequentially consistent collections, write operations are restricted to the leader replica. To enforce this single-writer guarantee, a function must declare its intent to perform writes in its metadata. This allows our scheduler to grant concurrent read permissions to multiple functions, but only a single function can write at a time on the leader.

Figure 4.4 illustrates the complete invocation lifecycle for an authenticated user, including metadata lookup, data discovery, and function execution. The diagram depicts a cold-start scenario with cache misses on both the client and the *Ermes Nodes*. In this example, the initial request is handled by a Tier 1 node that stores the required metadata, while the target data collection resides on its parent node in Tier 2.

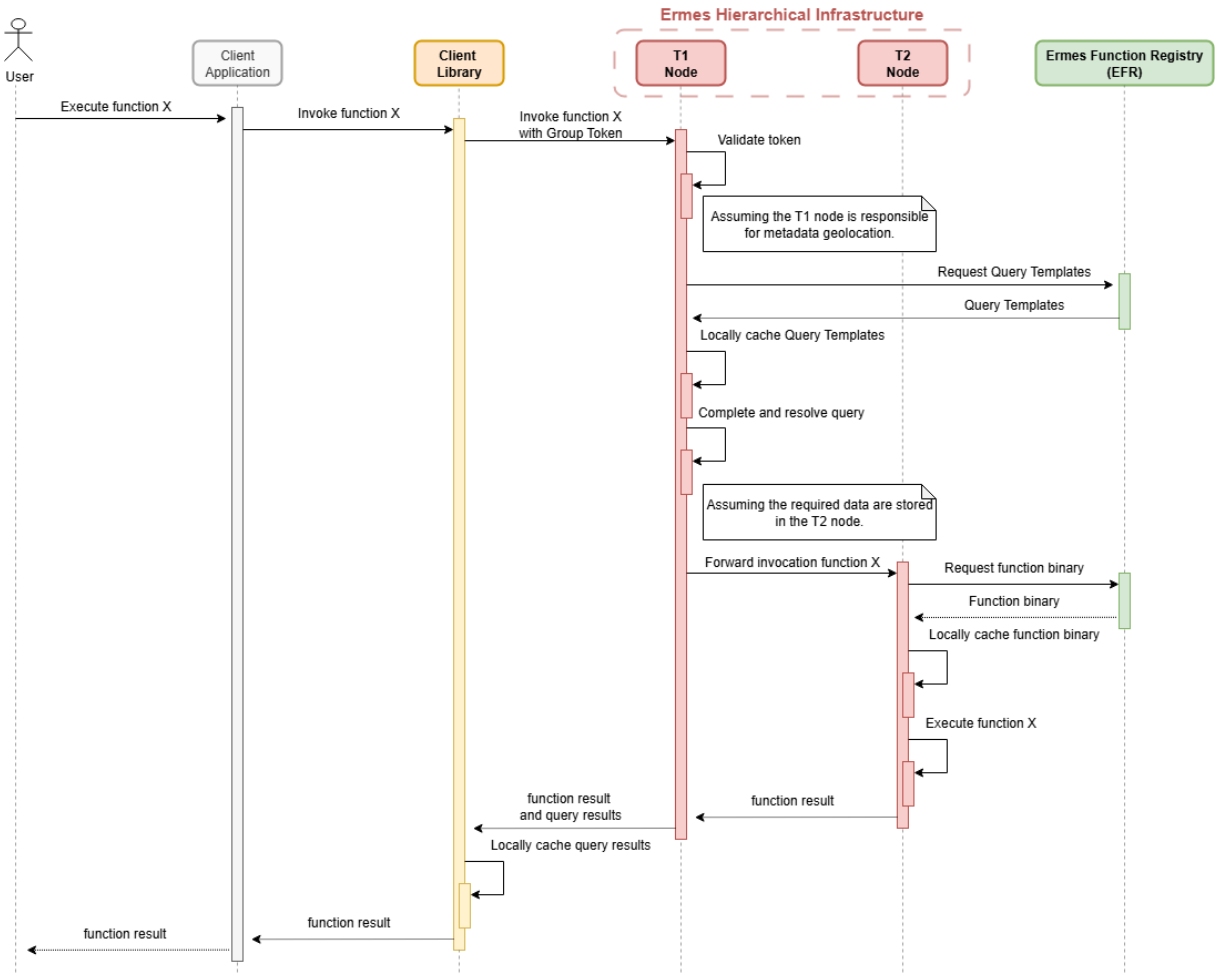


Figure 4.4: Function invocation lifecycle.

Update Propagation and Consolidation

After a function completes, our framework consolidates the updates buffered in the function-private write-back cache and propagates them to other replicas across the network. This "write-buffering" approach ensures that all modifications made within a single function invocation are applied atomically. The propagation itself follows a two-phase, asynchronous "up-and-down" protocol rooted at the execution node:

1. **Upward Propagation:** The execution node sends the updates to its parent. The updates continue to propagate recursively up the hierarchy, ensuring eventual state consolidation at the cloud layer.

2. **Downward Propagation:** From the cloud, or the lowest common ancestor of all replicas, the updates are then propagated down into the subtrees where other replicas are known to reside.

This protocol leverages the network topology to efficiently disseminate state changes. The specific semantics of this propagation, however, are dictated by the collection's consistency model.

For a sequentially consistent collection, write operations are only performed on the leader replica. The execution node commits the buffered writes as an atomic batch to the leader. The function's execution is considered complete only after the leader synchronously acknowledges the commit.

For an eventually consistent collection, the execution node directly initiates the asynchronous propagation of the buffered writes using the hierarchical protocol. Because writes can occur on any replica, a conflict resolution strategy is necessary to ensure that all replicas eventually converge. While established solutions such as Conflict-free Replicated Data Types (CRDTs) [56] offer strong guarantees, they can introduce significant implementation complexity and performance overhead. To balance correctness and efficiency, we opted for a Last-Write-Wins (LWW) policy. In our implementation, we resolve conflicts by using operation timestamps to determine the most recent update, with unique node identifiers serving as a tie-breaker to establish a total order.

5 | Centralized Solution for Data Placement

The Ermes framework integrates function execution with data collection management, enabling function invocations across any network node in a stateful environment governed by different consistency models, as detailed in Section 4.5.3. To optimize system performance, we introduce an optimization component responsible for the *collection placement strategy*, designed to minimize execution time and end-to-end latency. While such a strategy may be realized through either centralized or decentralized mechanisms, this chapter focuses on a centralized formulation to characterize the movement of data and collections. The subsequent chapter will address a decentralized policy designed for distributed environments.

We provide a formal foundation for this problem by extending the Integer Linear Programming (ILP) formulation for function offloading proposed by Nardelli et al. [40]. We adapt this model to the specific architectural requirements of Ermes, particularly the scheduling policies described in Section 4.5, which allow functions to be dynamically dispatched across the network. By formulating the problem as an ILP, we identify a globally optimal placement strategy that minimizes a cost function accounting for both function execution time and node storage consumption. This model serves as a theoretical benchmark, allowing us to precisely define the problem space and analytically evaluate the trade-offs between consistency models under the ideal assumption of a centralized orchestrator with a complete view of the system state.

Our formulation follows a modular design. We first establish a baseline model for environments without data replication, where functions and their required data must be co-located. We then extend this foundation to incorporate replication according to the consistency strategies defined in Section 4.4.1, specifically **Sequential Consistency** and **(Strong) Eventual Consistency**, by refining function execution constraints and data access overheads. Before detailing the mathematical formulation, we outline the fundamental assumptions that bound our models.

5.1. Assumptions

In order to facilitate our definition of the models, we establish the following simplifying assumptions:

- Data is grouped into collections, as defined in Section 4.4, and are treated as atomic units for placement and migration.
- We do not explicitly model the *Group Token Provider* (GTP) or the *Ermes Function Registry* (EFR). We assume functions possess intrinsic knowledge of their required collections and that function binaries are pre-distributed across all nodes.
- Network transport channels can be considered symmetric.
- Data retrieval from a local storage can be considered negligible.
- In presence of data replication, we do not model the complete destruction of collections.
- Functions that access a single collection need to be executed on the same node where the collection is present, without performing remote calls.
- Functions accessing only eventually consistent collections must execute on a node hosting at least one of the required collections, others can be retrieved remotely.
- Functions are restricted to accessing at most one sequentially consistent collection. Execution occurs on either the leader or a replica, depending on whether the function performs write or read-only operations.
- Functions that access both eventually and sequentially consistent collections must prioritize placement based on the sequentially consistent collections, with remote executions on eventual collections if needed.

Our objective is to determine the optimal distribution of collections and functions to minimize the total execution time perceived by the client.

5.2. Definitions

Let F be the set of functions, where each $f \in F$ is characterized by a memory demand $m_f \in \mathbb{R}$ and an execution time $r_f \in \mathbb{R}$ on a reference architecture.

Let N be the set of nodes, where each node $i \in N$ has a memory capacity $M_i \in \mathbb{N}$, storage capacity $S_i \in \mathbb{N}$, and a computational speedup factor s_i. Nodes $i, j \in N$ are connected

with non-negligible network latency $d_{i,j}$ and bandwidth $b_{i,j}$. We assume a symmetric network where $d_{i,j} = d_{j,i}$ and $b_{i,j} = b_{j,i}$.

Let K be the set of data collections, where each $k \in K$ has a size $l_k \in \mathbb{N}$. Each function $f \in F$ accesses a subset of collections $K_f \subseteq K$.

We partition K into two disjoint subsets based on consistency requirements:

$$K = K^S \cup K^E, \quad K^S \cap K^E = \emptyset \tag{5.1}$$

where K^S and K^E represent sequentially and eventually consistent collections, respectively. Per our assumptions on function accesses, $|K_f^S| \leq 1$ for all f, and we denote this single collection as $k_f^{(S)}$.

For functions accessing collections requiring Sequential Consistency, we require their declarations to specify whether data access is read-only or read-write, enabling us to construct the read set (R_f) and write set (W_f) for each function f. We can further distinguish between read-only functions (F_R) and functions that perform at least one write operation (F_W), where $F = F_R \cup F_W$.

The total time $T_{i,j}^f$ to execute function f on node j, given a request from node i, is defined as:

$$T_{i,j}^f = T_{f,(i,j)}^{IO} + T_{f,j}^{exec} + T_{f,j}^{data} \tag{5.2}$$

The I/O overhead $T_{f,(i,j)}^{IO}$ accounts for input size $l_{(f,I)}$ and output size $l_{(f,O)}$:

$$\begin{aligned} T_{f,(i,j)}^{IO} = T_{f,(i,j)}^{IN} + T_{f,(i,j)}^{OUT} &= \left(d_{i,j} + \frac{l_{(f,I)}}{b_{i,j}}\right) + \left(d_{j,i} + \frac{l_{(f,O)}}{b_{j,i}}\right) \\ &= 2d_{i,j} + \frac{l_{(f,I)} + l_{(f,O)}}{b_{i,j}} \end{aligned} \tag{5.3}$$

The execution time on node j is $T_{f,j}^{exec} = r_f / s_j$, where s_j is the computational speedup factor of node j relative to a reference architecture. The term $T_{f,j}^{data}$, is comprised by the time to access data locally (which we can assume negligible) and the time to retrieve the eventually consistent collections:

$$T_{f,j}^{data} = \sum_{k \in K_f^E} T_{\alpha,j}^{mig(k)} \tag{5.4}$$

where the migration time $T_{\alpha,j}^{mig(k)}$ of a single collection k from source α to j is:

$$T_{\alpha,j}^{mig(k)} = T_{j,\alpha}^{req} + T_{\alpha,j}^{ret} \tag{5.5}$$

Here, $T_{j,\alpha}^{req}$ is the time to access the node containing the collection remotely, which involves a data payload consisting of a collection ID and the operation to be performed, which we approximate using the network latency $d_{j,\alpha}$. $T_{\alpha,j}^{ret}$ represents the time taken to transfer the collection locally for caching purposes, defined as $d_{\alpha,j} + l_k/b_{\alpha,j}$. The term $T_{\alpha,j}^{ret}$ finally becomes:

$$T_{\alpha,j}^{mig(k)} = \begin{cases} 0 & \text{if } \alpha = j \\ 2d_{j,\alpha} + \frac{l_k}{b_{\alpha,j}} & \text{if } \alpha \neq j \end{cases} \tag{5.6}$$

Table 5.1 summarizes the notation used.

Symbol	Description
F	Set of functions
m_f	Memory utilized by $f \in F$
r_f	Execution time on a reference architecture of function $f \in F$
N	Set of nodes
M_i	Memory capacity of node $i \in N$
S_i	Storage capacity of node $i \in N$
s_i	Computation speed-up of node $i \in N$
$d_{i,j}$	Latency between nodes $i, j \in N$
$b_{i,j}$	Bandwidth between nodes $i, j \in N$
K	Set of all collections
l_k	Size of collection $k \in K$
F_W	Set of all functions $f \in F$ with a non-empty write set W_f
F_R	Set of all functions $f \in F$ with an empty write set W_f
$T_{i,j}^{f}$	Total time taken for function $f \in F$ to be executed on node $j \in N$ originating from node $i \in N$
$T_{i,j}^{mig(k)}$	Time taken for collection $k \in K$ to be migrated from node $i \in N$ to node $j \in N$

Table 5.1: Main notation adopted in the mathematical models.

5.3. No Replication

We first define a baseline model without replication, where each function and its required collection must reside on the same node. This formulation establishes the objective function and core decision variables used in subsequent models.

We define the initial allocation of collections using a set of constant binary parameters $\bar{x}_i^{(k)} \in \{0,1\}$, where $\bar{x}_i^{(k)} = 1$ iff collection $k \in K$ is initially allocated on node $i \in N$, and 0 otherwise. We assume that each collection is initially located on exactly one node: $\sum_{i \in N} \bar{x}_i^{(k)} = 1, \forall k \in K$.

We employ a similar approach for functions, introducing a binary parameter $\bar{w}_i^{(f)}$ which indicates that function f has been called from node i iff $\bar{w}_i^{(f)} = 1$.

The decision variables for our baseline model are:

- $x_j^{(k)} \in \{0,1\}$: 1 if collection k is finally placed on node j.
- $w_j^{(f)} \in \{0,1\}$: 1 if function f is executed on node j.
- $z_{i,j}^{(f)} \in \{0,1\}$: 1 if a request for f from node i is routed to node j.

The function migration variable $z_{i,j}^{(f)}$ is directly related to the initial and final locations $\bar{w}_i^{(f)}$ and $w_j^{(f)}$. Specifically, function f is migrated from node i to node j $(i \neq j)$ if and only if it was initially on node i and is finally on node j. Mathematically, $z_{i,j}^{(f)}$ represents the logical AND between $\bar{w}_i^{(f)}$ and $w_j^{(f)}$, which can be expressed through the following constraints:

$$\begin{aligned} z_{i,j}^{(f)} &\leq \bar{w}_i^{(f)} \\ z_{i,j}^{(f)} &\leq w_j^{(f)} \\ z_{i,j}^{(f)} &\geq \bar{w}_i^{(f)} + w_j^{(f)} - 1 \end{aligned} \tag{5.7}$$

Without replication, the execution time of a single function does not include a term for remotely requesting data; hence, $T_{f,j}^{data} = 0$. However, we must account for the time incurred when transporting the request only if the function call originates from a node i different from the execution node j.

Consequently, Equation 5.2 becomes:

$$T_{i,j}^{f} = \left(2d_{i,j} + \frac{l_{(f,I)} + l_{(f,O)}}{b_{i,j}}\right)(1 - \bar{w}_j^{(f)}) + \frac{r_f}{s_j} \tag{5.8}$$

The total operational cost for all function requests is the sum of the execution times of each function:

$$\mathcal{F} = \sum_{f \in F} \sum_{i,j \in N} \frac{T_{i,j}^{f}}{T_{max}} z_{i,j}^{(f)} \tag{5.9}$$

Here we introduce a normalization term T_{max} defined as the sum of the worst-case execution times for each function: $T_{max} = \sum_{f \in F} \max_{i,j \in N} T_{i,j}^{f}$.

The **objective function** is to minimize the total cost function execution cost:

$$\min \mathcal{F} \tag{5.10}$$

We must include the following constraints for the ILP:

$$\sum_{k \in K} l_k x_j^{(k)} \leq S_j \qquad \forall j \in N \quad \text{(Storage Capacity Constraint)} \tag{5.11}$$

$$\sum_{f \in F} m_f w_j^{(f)} \leq M_j \qquad \forall j \in N \quad \text{(Memory Capacity Constraint)} \tag{5.12}$$

$$\sum_{j \in N} x_j^{(k)} = 1 \qquad \forall k \in K \quad \text{(Each Collection in One Final Location)} \tag{5.13}$$

Finally we can formalize our assumption that a function executes on the same node where the collection is located when solving the problem by explicitly stating that their final positions must coincide:

$$w_i^{(f)} = x_i^{(k_f)} \qquad \forall i \in N, \forall f \in F \tag{5.14}$$

5.4. Eventual Consistency

For collections requiring eventual consistency, we can now relax our baseline requirements to permit remote data accesses. For this analysis, we assume $K^S = \emptyset$ and $K^E \neq \emptyset$. By leveraging the scheduling properties of Ermes, we need to invoke a function on the nearest node that contains at least one replica of a collection in K_f^E, and from that node, it may perform remote data calls.

We introduce a new term which represents the time taken to perform remote operations on the closest replica of each eventually consistent collection accessed by function f:

$$T_{*,j}^{f,E} = \sum_{k \in K_f^E} \min_{\alpha \in N} (2d_{j,\alpha} + \frac{l_k}{b_{\alpha,j}})(1 - \bar{x}_j^{(k,E)}) \tag{5.15}$$

It is important to note that while this term appears nonlinear, it can be computed during model construction since it applies only to explicit variables.

With the introduction of this term, we can now consider the full definition of function execution time from Equation 5.2 where $T_{f,j}^{data} = T_{*,j}^{f,E}$. Its definition in its extended form becomes:

$$T_{i,j}^{f,E} = \left(2d_{i,j} + \frac{l_{(f,I)} + l_{(f,O)}}{b_{i,j}}\right)(1 - \bar{w}_j^{(f)}) + \frac{r_f}{s_j} + T_{*,j}^{f,E} \tag{5.16}$$

Here, we again consider the cost of moving functions and data only if they were not initially present on node j.

Our eventual consistency model keeps the same function execution cost of the baseline, originally defined in Equation 5.10, but in a replicated system an objective function focused solely on minimizing client-perceived latency could lead to degenerate solutions. For example, the solver might determine that replicating a collection on every node is optimal, as this would make all data accesses local. While theoretically minimizing latency within the model's scope, such a strategy is practically infeasible due to significant unmodeled operational costs. These costs include the network overhead of propagating updates to all replicas, the computational cost of maintaining consistency protocols, and the raw expense of storage.

To account for these implicit costs without modeling the full complexity of the replication strategy, we introduce a regularization term $\mathcal{C}$ responsible for penalizing the total number and size of collection replicas across the system:

$$\mathcal{C} = \sum_{i \in N} \sum_{k \in K} \frac{l_k}{l_{max}} x_i^{(k,E)} \tag{5.17}$$

where l_{max} is a normalization term defined as the sum of the size of each collection if it were replicated on every node: $l_{max} = |N| \cdot \sum_{k \in K} l_k$. This term serves as a proxy for the underlying cost of maintaining each replica, forcing the optimizer to find a meaningful trade-off between reducing data access latency and minimizing the system-wide burden of replication.

Our new **objective function** now comprises of two terms, the operational cost for all function requests $\mathcal{F}$, defined in Equation 5.9, and the new regularization term $\mathcal{C}$:

$$\min \mathcal{F} + \mathcal{C} \tag{5.18}$$

Moreover we need to relax the original constraints on the uniqueness of a collection and on the scheduling of each function, defined in constraints 5.13 and 5.14.

As specified by our Ermes scheduling policy, a function f that requires eventually consistent collections can only be executed on a node i if that node already contains at least one of the required collections. This introduces a conditional relationship between the function placement decision variable ($w_i^{(f)}$) and the data placement decision variables ($x_i^{(k)}$).

To correctly formulate this if-else constraint, we introduce two distinct linear transformations [24] that facilitate their conversion into standard form:

Proposition 5.1. *Let $x, y \in \{0, 1\}$ be decision variables of an optimization problem, and a constraint defined as:*

$$x = 1 \implies y = 1$$

Then the constraint can be rewritten as:

$$x \leq y \tag{5.19}$$

Next, we introduce a constraint that defines the transformation of a constraint utilizing a linear operator $g(\cdot)$:

Proposition 5.2. *Let $x, y \in \{0, 1\}$ be decision variables of an optimization problem, and a constraint defined as:*

$$x = 1 \implies g(y) \leq b$$

where $b \in \mathbb{R}$ and $g(y)$ is a linear transformation of variable y.

Then the constraint can be rewritten as:

$$g(y) \leq b + \mathcal{M}(1 - x) \tag{5.20}$$

where $\mathcal{M}$ is a sufficiently large constant.

For constraints defined as:

$$x = 1 \implies g(y) \geq b$$

the transformation becomes:

$$g(y) \geq b - \mathcal{M}(1 - x) \tag{5.21}$$

For equality statements, the resulting constraint is the combination of Equations 5.20 and 5.21.

Using the principle from Proposition 5.2, we can now formally express the scheduling constraint that a function must execute on a node containing at least one of its required replicas as follows:

$$\sum_{k \in K_f^E} \bar{x}_i^{(k,E)} \geq 1 - \mathcal{M}(1 - w_i^{(f)}) \qquad \forall f \in F, \forall i \in N \tag{5.22}$$

$$x_i^{(k,E)} \geq \bar{x}_i^{(k,E)} \qquad \forall k \in K^E \forall i \in N \tag{5.23}$$

where we also constraint each collection to leave a replica in its initial position.

Finally, the uniqueness constraint on the placement variable $x_j^{(k,E)}$ can be removed, as multiple replicas are now allowed. However, we still require that at least one replica per collection is present in the system at the end of the execution of the optimization model:

$$\sum_{j \in N} x_j^{(k,E)} \geq 1 \qquad \forall k \in K \tag{5.24}$$

5.5. Sequential Consistency

We now address the problem of sequential consistency, assuming for this section that $K_f^E = \emptyset$ and $K_f^S \neq \emptyset$. Ermes assumes that a function accesses at most one data collection requiring strong consistency, which we represent as a single collection $k_f \in K^S \forall f \in F$.

Following our definition of read and write set of a function in Section 5.2, if the write set W_f of a function is empty, the invocation can be routed to the nearest replica. Otherwise, for functions with non-empty write sets, execution must occur on the leader node of the replication protocol.

To formalize the model, we first define the representation of leaders and replicas. We introduce a binary variable $\lambda_i^{(k,S)}$ to identify node i as the leader for collection k.

Consequently, we define a node i that contains a non-leader replica of the data as:

$$x_i^{(k,S)} = 1 \wedge \lambda_i^{(k,S)} = 0 \tag{5.25}$$

Following the structure of the formulation in Equation 5.18, we define the **objective function** as the sum of total function execution time and the penalization term of the number of replicas. Our model is therefore expressed as:

$$\min(\mathcal{F}^{leader} + \mathcal{C}^{leader}) + (\mathcal{F}^{replica} + \mathcal{C}^{replica}) \tag{5.26}$$

The components for function execution time and data movement cost are analogous to Equations 5.9 and 5.17, with the time taken to execute a function being defined by Equation 5.8. The primary distinction lies in the placement rules: leaders execute all functions in F_W and may also execute functions from F_R, whereas non-leader replicas are restricted to executing only functions from F_R. The objective function is then explicitly defined as:

$$\begin{aligned} \min \sum_{f \in F_W} \sum_{i,j \in N} \frac{T_{i,j}^f}{T_{max}^{(S)}} z_{i,j}^{(f)} + \sum_{i \in N} \sum_{k \in K^S} \frac{l_k}{l_{max}^{(S)}} \lambda_i^{(k,S)} + \\ \sum_{f \in F_R} \sum_{i,j \in N} \frac{T_{i,j}^f}{T_{max}^{(S)}} z_{i,j}^{(f)} + \sum_{i \in N} \sum_{k \in K^S} \frac{l_k}{l_{max}^{(S)}} x_i^{(k,S)} (1 - \lambda_i^{(k,S)}) \end{aligned} \tag{5.27}$$

Since $F_W \cup F_R = F$, we can combine the summations over the function sets:

$$\min \sum_{f \in F} \sum_{i,j \in N} \frac{T_{i,j}^f}{T_{max}^{(S)}} z_{i,j}^{(f)} + \sum_{i \in N} \sum_{k \in K^S} \frac{l_k}{l_{max}^{(S)}} \lambda_i^{(k,S)} + \sum_{i \in N} \sum_{k \in K^S} \frac{l_k}{l_{max}^{(S)}} x_i^{(k,S)} (1 - \lambda_i^{(k,S)}) \tag{5.28}$$

Finally, we consolidate the data movement costs for leaders and replicas into a single

term, given $\lambda_i^{(k,S)} = 1 \implies x_i^{(k,S)} = 1$:

$$\min \sum_{f \in F} \sum_{i,j \in N} \frac{T_{i,j}^f}{T_{max}^{(S)}} z_{i,j}^{(f)} + \sum_{i \in N} \sum_{k \in K^S} \frac{l_k}{l_{max}^{(S)}} x_i^{(k,S)} \tag{5.29}$$

where the normalization factors are the same as the eventual consistency model.

We observe that the introduction of the leader does not alter the fundamental formulation of the problem but significantly impacts the introduced constraints.

To complete the problem formulation, we must define the constraints. Specifically, we need to model the relationship between the leader node and the set of replicas for a given collection, which can be written as follows:

$$\lambda_i^{(k,S)} \leq x_i^{(k,S)} \qquad \forall i \in N, \forall k \in K^S \tag{5.30}$$

$$\sum_{i \in N} \lambda_i^{(k,S)} = 1 \qquad \forall k \in K^S \tag{5.31}$$

$$\sum_{i \in N} x_i^{(k,S)} \geq 1 \qquad \forall k \in K^S \tag{5.32}$$

Here, we still specify that there must be at least one replica per collection, similar to Equation 5.24, with the uniqueness constraint on the leader.

Next, we explicitly formulate the Ermes scheduling policy by stating that if function f performs only read operations (i.e., $W_f = \emptyset$), it can execute on either the leader or any replica. If the function performs write operations, it must execute exclusively on the leader:

$$w_i^{(f)} \leq x_i^{(k_f,S)} \qquad \forall i \in N, \forall f \in F_R \tag{5.33}$$

$$w_i^{(f)} \leq \lambda_i^{(k_f,S)} \qquad \forall i \in N, \forall f \in F_W \tag{5.34}$$

$$\sum_{i \in N} w_i^{(f)} = 1 \qquad \forall f \in F \tag{5.35}$$

Here, we use our assumption that $|K_f| = 1$ for all $f \in F$, which implies that a single collection k_f can be assigned to each function f.

5.6. Combining the Consistency Models

We now integrate the Eventual Consistency model defined in Section 5.4 with the Sequential Consistency model defined in Section 5.5. This integration is achieved by incorporating the data access cost for eventually consistent collections into the function execution time term of the objective function. The resulting comprehensive model is:

$$\min \sum_{f \in F} \sum_{i,j \in N} \left((2d_{i,j} + \frac{l_{(f,I)} + l_{(f,O)}}{b_{i,j}})(1 - \bar{w}_j^{(f)}) + \frac{r_f}{s_j} + T_{*,j}^{f,E} \right) \cdot \frac{1}{T_{max}} \cdot z_{i,j}^{(f)} + \sum_{i \in N} \sum_{k \in K^S} \frac{l_k}{l_{max}^{(S)}} x_i^{(k,S)} + \sum_{i \in N} \sum_{k \in K^E} \frac{l_k}{l_{max}^{(E)}} x_i^{(k,E)} \tag{5.36}$$

The decision variables for the final optimization problem are the binary variables for eventually consistent data placement ($x_i^{(k,E)}$), sequentially consistent data placement ($x_i^{(k,S)}$), leader assignment ($\lambda_i^{(k,S)}$), and function placement ($z_{i,j}^{(f)}$), and the constraints remain the same for both models, with the only clarification that function which access collections of different consistencies must prioritize the sequential collection.

5.7. Model Complexity and Scalability

The model dimension is governed by three dominant factors, the number of decision variables of the problem, the number of constraints and the dimension of the objective function.

We identified an opportunity to optimize problem construction by significantly reducing the number of variables of the model. This section formalizes the methodology for variable reduction, identifying which variables can be retained or removed, and how these changes impact the asymptotic scaling of the problem. We structure this discussion into two subsections: one focusing solely on the no-replication model and another covering both eventual and sequential consistency models, given that the reduction techniques applied to these models share a similar nature.

5.7.1. No Replication

The number of decision variables for the baseline model described in Section 5.3 depends on three primary factors: the number of functions $|F|$, the number of collections $|K|$, and the number of nodes $|N|$ within the system.

Specifically, for each node i, we introduce a decision variable representing the final position of function f, denoted $w_i^{(f)}$, and another for collection k, denoted $x_i^{(k)}$. Additionally, for the functions, we require displacement variables $z_{i,j}^{(f)}$ for the movement from node i to node j.

Consequently, the total number of variables in the problem is expressed as:

$$\Theta(|F| \cdot |N| + |K| \cdot |N| + |F| \cdot |N|^2) = O\big(|F| \cdot |N|^2\big) \tag{5.37}$$

Given the co-location constraint in Equation 5.14, the decision variables $w_i^{(f)}$ are fully determined by the collection placement variables $x_i^{(k)}$. This allows us to eliminate the $w_i^{(f)}$ variables entirely by substituting $x_i^{(k_f)}$ in their place throughout the model.

Furthermore, the function migration variable $z_{i,j}^{(f)}$ is defined as the logical AND between the parameter $\bar{w}_i^{(f)}$ and the decision variable $w_j^{(f)}$. After substituting for $w_j^{(f)}$, the term $z_{i,j}^{(f)}$ is equivalent to $\bar{w}_i^{(f)} \wedge x_j^{(k_f)}$. Since $\bar{w}_i^{(f)}$ is a known constant (a parameter), we can replace the variables $z_{i,j}^{(f)}$ in the objective function directly. The term $\sum_{i,j \in N} T_{i,j}^f z_{i,j}^{(f)}$ simplifies to $\sum_{j \in N} T_{i_0,j}^f x_j^{(k_f)}$, where i_0 is the known origin node of the request for function f (i.e., where $\bar{w}_{i_0}^{(f)} = 1$).

This substitution reduces the set of decision variables to only $x_i^{(k)}$. Consequently, the total number of variables decreases from $O(|F| \cdot |N|^2)$ to $\Theta(|K| \cdot |N|)$, fundamentally changing the problem's dimensionality from quadratic to linear in the number of nodes.

5.7.2. Replication

For models incorporating replication, the co-location constraint is relaxed, and thus the function placement variables $w_j^{(f)}$ must remain as independent decision variables. However, the function migration variables $z_{i,j}^{(f)}$ can still be eliminated to reduce the problem's dimensionality.

Initially, the number of decision variables for the eventual consistency model remains consistent with the non-optimized no-replication model, as defined in 5.37. For the sequential consistency model, additional variables are required to describe the position of the leader $\lambda_i^{(k,S)}$. This increases the total number of variables to:

$$\Theta(|F| \cdot |N| + 2 \cdot |K| \cdot |N| + |F| \cdot |N|^2) = O\big(|F| \cdot |N|^2\big) \tag{5.38}$$

Here we cannot directly remove the function movement variable $z_{i,j}^{(f)}$, but we can achieve

a further reduction in the number of variables by leveraging its relationship with $w_i^{(f)}$. Specifically, $z_{i,j}^{(f)}$ is defined by the logical relationship $z_{i,j}^{(f)} \iff \bar{w}_i^{(f)} \wedge w_j^{(f)}$. Instead of introducing $z_{i,j}^{(f)}$ as an auxiliary variable governed by linearization constraints (as in Eq. 5.7), we can directly substitute its logical equivalent into the objective function.

Since $\bar{w}_i^{(f)}$ is a known parameter, we can replace $z_{i,j}^{(f)}$ with the linear expression $\bar{w}_i^{(f)} \cdot w_j^{(f)}$. The objective function term becomes $\sum_{f,i,j} \frac{T_{i,j}^f}{T_{max}} (\bar{w}_i^{(f)} \cdot w_j^{(f)})$. This reformulation is linear because it involves the product of a constant ($\bar{w}_i^{(f)}$) and a decision variable ($w_j^{(f)}$).

Using this transformation, the new total number of variables for the two models yield the following complexities: for eventual consistency:

$$\Theta(|F| \cdot |N| + |K| \cdot |N|) = \Theta\Big((|F| + |K|) \cdot |N|\Big) \tag{5.39}$$

and for sequential consistency:

$$\Theta(|F| \cdot |N| + 2 \cdot |K| \cdot |N|) = \Theta\Big((|F| + 2 \cdot |K|) \cdot |N|\Big) \tag{5.40}$$

In summary, the final complexity of the model can be transformed from quadratic to linear by applying the correct transformations to the variables.

6 | Distributed Solution for Data Placement

The Integer Linear Programming (ILP) model presented in the previous chapter defines an optimal, centralized solution to the data placement problem. However, a centralized approach is impractical for a FaaS platform deployed across the Edge-Cloud continuum. Such an approach would require capturing a consistent global view of the system state, including collection locations and data access patterns, potentially via distributed snapshot algorithms [10], before aggregating this information at a central orchestrator. Furthermore, a distributed locking protocol [19] would be necessary to suspend system execution while migrating collections to their optimal locations. These processes are computationally intensive and incur prohibitive network overhead. Moreover, as analyzed in Section 5.7, the ILP problem can become computationally expensive to solve at scale. These factors are particularly problematic in dynamic environments characterized by mobile clients and shifting workloads, where a global view would likely become stale by the time a placement decision is computed, rendering the solution suboptimal.

These limitations motivate the need for a practical, decentralized solution where nodes make independent placement decisions based on limited, partial knowledge of the system state. To formalize this distributed process, we introduce the concept of an *anchor node*. For any given data collection, an anchor node is a node that hosts a replica and possesses the necessary local state information to execute the placement logic for that replica.

Prior to presenting our heuristic-based solution to the data placement problem, we define the scope of knowledge available to each node regarding the network configuration and current data locations. As detailed in Section 4.3, Ermes exploits a hierarchical network topology. Within this topology, a node is aware of its direct children and maintains an aggregated summary of the data accessible through its descending links. For each descending path, a node knows whether a data item is reachable, but not its exact location within the corresponding subtree. Conversely, if a data item is not found in any of its subtrees, the node infers that it must reside at an upstream location.

Our primary objective for the data placement problem is to minimize the user-perceived latency of function invocations. To this end, our solution aims to store data in close proximity to the users accessing it. Our framework supports two distinct consistency strategies for collections, as detailed in Section 4.4.1, which entail different replication behaviors. **Eventual Consistency** inherently enables aggressive replication strategies, as it allows both read and write operations to be served by any available replica. Since update propagation and conflict resolution are performed asynchronously, the associated synchronization latency does not impact the critical path of user requests. Consequently, these properties allow us to treat data placement as a **distributed caching problem**.

Conversely, we implement **Sequential Consistency** collections using a single-leader strategy. In this configuration, all write requests are directed to a designated leader replica, which subsequently propagates updates to follower replicas asynchronously. Read-only functions, by contrast, can be served by any available replica. Consequently, our objective is to strategically place the leader and its followers to minimize the average user-perceived latency for both read and write operations. This requirement introduces a complex state space, rendering simple caching heuristics insufficient. To address this complexity, we formulate the data placement problem using a **physical model**.

6.1. Distributed Caching Model

When a function invocation requires a data collection not present at the entry node, the request is routed through the network hierarchy to a remote anchor node hosting the required data. We refer to this approach as the **always-copy policy**, as it exploits the aggressive replication capabilities of the Eventual Consistency model to minimize latency for subsequent accesses. Specifically, upon completion of a function execution accessing a remote collection, the anchor node decides to asynchronously instantiate a new replica on the node immediately preceding the execution node in the routing path (i.e., one hop closer to the user), provided that the node has sufficient memory.

Each node manages its local storage using a Least Recently Used (LRU) eviction policy. When a node requires additional memory, it evicts the least recently used replica by propagating it to its parent in the network hierarchy. The parent node attempts to store the incoming collection; if it lacks sufficient capacity, it triggers its own eviction process according to the LRU policy to accommodate the new data. This cascading upward propagation continues until the replica reaches a node with adequate free memory or reaches the Cloud, which we assume possesses unlimited storage capacity. If a propagated collection arrives at a node that already hosts a replica of the same data, the evicted copy

is discarded to avoid redundancy.

A key advantage of this caching-based strategy is that it eliminates the need for anchor nodes to maintain access pattern information for collections under Eventual Consistency. Consequently, this design significantly reduces the network and memory overhead associated with our data placement solution.

6.2. The Physical Model

The replica placement problem for Sequential Consistency collections entails a significantly larger state space compared to Eventual Consistency. To address this complexity, we initially focus on a single data collection, as the placement for each collection can be optimized independently. We reduce the dimensionality of the problem by constraining the placement search space to the subtree rooted at the collection's *Common Ancestor*. We define the Common Ancestor of a set of nodes as the root of the minimal subtree encompassing all nodes in that set. Specifically, if a set of nodes serves as entry points for users accessing a collection, we restrict the placement of replicas and Anchor Nodes to the subtree rooted at the Common Ancestor of these entry points. Consequently, we assume that the Common Ancestor maintains aggregated knowledge of the access patterns for the collection.

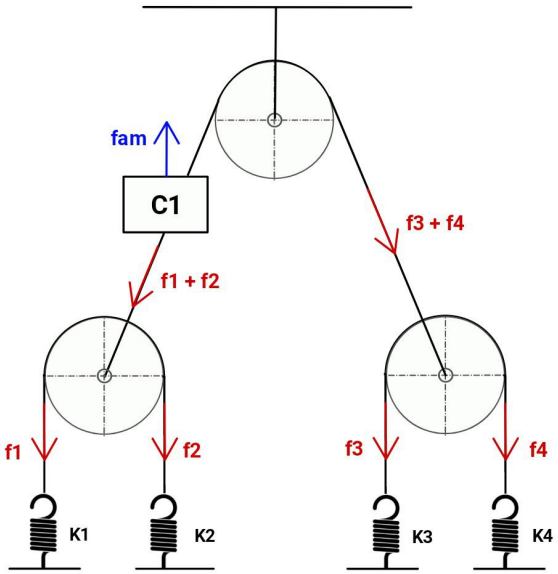


Figure 6.1: Physical representation of a 3-node subtree.

We model this constrained search space as a physical system, analogous to a system

of ropes and pulleys as depicted in Figure 6.1, where data replicas are influenced by competing forces. To further reduce problem dimensionality, we restrict our analysis to a binary tree structure. In this analogy, each network node corresponds to a fixed point associated with the fulcrum of a pulley, while data collections are conceptually positioned along the ropes, moving freely until an equilibrium position is reached. While our model addresses a distinct problem and employs a different physical formulation, it is conceptually inspired by the approach introduced in "Network-Aware Operator Placement for Stream-Processing Systems" [48].

We consider two primary opposing factors influencing the placement of each replica.
First, we represent the user's objective to maintain data proximity as an elastic force. This force is proportional to the number of access requests and the data access latency (represented by geographical distance) between the user and the requested collection, analogous to the elongation of a spring from its resting position.

Second, we consider the network's objective to prevent node overload and preserve available space within node LRU caches. The stability of a collection's placement is influenced by its depth within the LRU, the size of the collection, and the memory pressure on the node. We represent this behavior using a repulsive buoyant force. Specifically, we model data items within a node's LRU as objects submerged in a constricted fluid container. The deeper an object is positioned within this fluid and the larger its volume (collection size), the higher its "potential energy." We leverage buoyant forces to model the interactions among collections co-located on the same node. Collections compete to maintain their state of motion within the fluid and to occupy portions of the node's LRU memory. Consequently, the placement and persistence of a collection depend on the memory consumed by other collections. This competitive interaction is captured by scaling the fluid density inversely with the free memory available on the node, thereby increasing the repulsive effect as the node approaches saturation.

To summarize, we define the following forces acting on the system:

- n **elastic forces**, one for each user access point, pulling replicas closer:

$$\vec{f}_{En} = -k \cdot \Delta\vec{v} \tag{6.1}$$

  where:

  - k is the number of access requests.
  - Δv is the network latency, corresponding to the distance between the user and

the closest replica.

- m **buoyant forces** , one for each of the m replicas, pushing them away from nodes with limited memory:

$$\vec{f}_{Bm} = \rho_{\mathrm{f}} \cdot V\vec{g} = \frac{\mu}{freeSize} \cdot size(c) \cdot \vec{g} \tag{6.2}$$

where, based on physical theory:

  - ρ_{f} is the fluid density.
  - V is the volume of the submerged object.
  - g is the gravitational acceleration.

and in our derived model:

  - $freeSize$ is the available memory on the node hosting the replica.
  - $size(c)$ is the size of the data collection considered.
  - μ, g are constant scaling factors.

We can determine the placement of the leader independently from the follower replicas. Although the leader also serves read requests, its placement is primarily driven by write traffic, whereas follower placement is driven by read traffic.
Accordingly, we decompose the total elastic force into read and write components:

$$\vec{f}_{En} = \vec{f}^{R}_{En} + \vec{f}^{W}_{En}$$

$$\vec{f}^{R}_{En} = -k^{R} \cdot \Delta\vec{v}$$

$$\vec{f}^{W}_{En} = -k^{W} \cdot \Delta\vec{v}$$

In this model, the optimal solution to the data placement problem under sequential consistency corresponds to the stable equilibrium of the physical system, i.e., the configuration that minimizes the total potential energy.
We express the elastic and hydrostatic potential energies as follows:

- **Elastic potential energy (read-only)**

$$E^{R}_{elastic} = \frac{1}{2} \cdot k^{R} \cdot \Delta v^{2} \tag{6.3}$$

- **Elastic potential energy (write-only)**

$$E^W_{elastic} = \frac{1}{2} \cdot k^W \cdot \Delta v^2 \tag{6.4}$$

- **Hydrostatic potential energy**

$$E_{hydrostatic} = -L = \rho_{\mathrm{f}} \cdot V \cdot g \cdot h = \frac{\mu}{freeSize} \cdot size(c) \cdot g \cdot h \tag{6.5}$$

  where:

  – h is the depth of the data collection in the LRU cache, which represents its eviction priority.

Our objective function is to minimize the total potential energy:

$$\min(E^W_{elastic} + \sum_n E^R_{elastic_n} + \sum_m E_{hydrostatic_m}) \tag{6.6}$$

In a physical system, an object moves spontaneously if that movement causes a decrease in its potential energy. Equilibrium is reached when no available moves can further reduce E_{total}. Accordingly, we compute each placement decision with the objective of balancing the forces currently acting on the collection.
Given a current placement, we consider a displacement advantageous if it contributes to force equilibrium. To this end, our algorithm optimizes two sets of variables:

1. The location of the leader node.
2. The number and locations of read-only follower replicas.

We minimize the first term of the objective function by positioning the leader at the location that balances the cumulative elastic forces derived from write requests.
The second and third terms, however, introduce a trade-off regarding the placement of follower replicas:

- Minimizing the second term ($E^R_{elastic_n}$) promotes the instantiation of multiple read-only replicas, specifically one for each significant source of read requests, to reduce access latency.
- In contrast, minimizing the third term ($E_{hydrostatic_m}$) favors creating fewer replicas and placing them higher in the network hierarchy, where nodes typically have more available memory and thus exert less memory pressure.

Therefore, our placement strategy seeks a configuration of minimum potential energy. This corresponds to a stable equilibrium in the continuous space that balances these competing objectives, thereby maximizing data locality for read operations while minimizing replica overhead and memory consumption.

6.3. Decision Policies for Sequentially Consistent Collections

Our physical model provides an intuitive, continuous description of the competing forces that govern optimal data placement in a hierarchical network structured as a binary tree. In our rope-and-pulley analogy, user access patterns create tension on ropes, and the cumulative tension on any node is the result of forces acting throughout its subtree, propagating recursively up the hierarchy.

In this section, we derive the practical, distributed decision policies for sequentially consistent replica placement in an N-ary tree based on this model. The central challenge is to translate the global equilibrium conditions into a discrete, decentralized algorithm where each node operates with only partial knowledge. We present the specific rules for placing the leader and follower replicas, detailing the rationale that links each local decision to a reduction in the system's total potential energy.

6.3.1. Model Notation and Symbols

Before introducing the decision rules derived from the model, we summarize the notation and symbols used throughout the analysis. Table 6.1 provides a concise overview of the variables, parameters, and functions that define the model.

Table 6.1: Main notation adopted in the physical model.

Symbol	Description
k	number of access requests
k^W	number of write accesses
k^W	number of read accesses
Δ	cumulative latency along a path in the network
δ	latency of a single arc in the topology

Symbol	Description
f_E	Elastic force, representing the user's interest in the data item under consideration
f_E^W	Write elastic force
f_E^R	Read elastic force
f_B	Buoyant force, representing the network's objective to prevent node overload

6.3.2. Placing the Leader

The placement of the leader replica is determined by a recursive algorithm. Each node aggregates the "write forces" exerted by clients within its subtree and propagates this cumulative information to its parent.

At each step in the placement algorithm, an anchor node hosting the leader evaluates whether to delegate the placement responsibility to one of its children. If the delegation condition is not met for any child, the anchor node hosts the leader itself.
When a node decides whether to delegate the leader to one of its children, the evaluation is based on the principle of potential energy reduction (Equation 6.6). In other words, the node assesses if a proposed move yields an effective reduction in the total potential energy.

We decompose the factors influencing this evaluation as follows:

- **Energy Reduction (Benefit):** The *elastic potential energy* decreases for all write requests originating within the subtree of the designated child i. The latency (represented as distance) between these clients and the collection decreases by a factor δv_i, and the aggregate force driving this reduction is denoted as $f_{Ei\in C}^W$.
- **Energy Increase (Cost):**
  1. The *elastic potential energy* increases for write requests originating from all other branches of the network. The latency between these clients and the collection increases by a factor δv_i. The work required to overcome this resistance is related to $k_j^W \cdot \delta v_i$.
  2. The *hydrostatic potential energy* increases at the destination node i due to local memory pressure.

Delegating is therefore advantageous if the energy benefit outweighs the energy cost.

When delegating, the anchor node communicates the cumulative write tension from all other parts of the network to the chosen child, providing updates as the tension changes. The child then recursively repeats this decision process.
An anchor node delegates leader placement to its i-th child if and only if the pull from that child's subtree outweighs the forces resisting the move.
This condition is met when:

$$f^W_{Ei\in C} - \sum_{j\in C:j\neq i} f^W_{Ej} - AG_{t-1} > f_{Bi} + \sum_{j\in C:j\neq i} k^W_j \cdot \delta v_i + T \qquad \forall i \in C \quad (6.7)$$

where:

- C is the set of **child nodes**.
- δv_i is the **latency** (or geographical distance) between the anchor node and the destination node. Therefore, the penalty incurred by moving the leader to the destination node is: $\sum_{j\in C:j\neq i} k^W_j \cdot \delta v_i$.
- f_{Ai} are the **Archimedean forces** in the destination node.
- T is a configurable **threshold constant** (optional).

The aggregated tension AG_t passed to the delegated child includes the tension from the anchor's parent (AG_{t-1}) and all sibling subtrees.
We compute it as follows:

$$AG_t = \sum_{j\in C:j\neq i} f^W_{Ej} + \sum_{j\in C:j\neq i} k^W_j \cdot \delta v_i + AG_{t-1} \qquad \forall i \in C \quad (6.8)$$

6.3.3. Placing Read-Only Replicas

We use a separate algorithm for placing read-only replicas. Unlike leader placement, this process creates new replicas rather than migrating a single instance. An anchor node can decide to create new replicas on multiple children simultaneously.
An anchor node retains its local replica as long as there exists residual read demand (i.e., elastic force) that is not satisfied by replicas located within its subtrees. Equivalently, the node preserves the replica as long as it continues to receive unsatisfied read requests from any of its descending links.
The decision to create a new read-only replica on the i-th child depends, as in the previous section 6.3.2, on the energy balance. The anchor node will create a new replica if the reduction in elastic potential energy (benefit) sufficiently outweighs the increase in hydrostatic potential energy (cost).

In this algorithm, the decision to replicate to a child depends only on the elastic force

from that child's subtree, not on forces from sibling subtrees. The buoyant force ($\vec{f}_{Ai}$) plays a more critical role here than in leader placement. Creating a new replica increases the total count m, which raises both the system's hydrostatic potential energy and its overall resource cost.
However, under our partial knowledge assumption, an anchor node does not know the total number of replicas m in the system. We therefore simplify the decision rule to depend only on local information. An anchor node replicates to child i if its elastic pull exceeds the local memory pressure by a given factor:

$$f^R_{Ei \in C} > f_{Bi} + t \qquad \forall i \in C \quad (6.9)$$

where t is a configurable threshold factor used to tune the sensitivity to memory pressure.

6.4. Relationship between the Mathematical Model and Physical Model

While a formal investigation into the equivalence of the two models is beyond the scope of this work, this section provides a qualitative comparison. We aim to highlight the conceptual parallels between the formulations and intuitively verify the physical model's adherence to the constraints imposed by the mathematical formulation
Both formulations share the objective of minimizing average user-side latency. However, they differ in the approach. The mathematical formulation addresses the general problem of joint function and data placement. In contrast, our physical model simplifies the problem by focusing on the optimal data placement at a specific instant, this is achieved by treating each data collection independently and considering an idealized scenario where each function accesses a single collection, for which an optimal placement is sought; functions accessing multiple collections may consequently experience suboptimal execution times. We then assume that functions are executed on the same node where their required data resides.

Despite their different perspectives, both formulations capture the same fundamental trade-off. They must balance the goal of bringing data closer to users against the costs of replication and increasing memory pressure. In our physical model, the elastic force ($\vec{f}_{En}$ in Eq. 6.1) maps directly to the total operational cost for all function requests ($\mathcal{F}$ in Eq. 5.9) in the mathematical model, representing the pull for data locality. Similarly, the buoyant force ($\vec{f}_{Bm}$ in Eq. 6.2) discourages solutions that concentrate data on a small subset of nodes and maps to the regularization term introduced in the mathematical

model ($\mathcal{C}$ in Eq. 5.17), which favors placing data higher in the hierarchy to conserve resources.

Furthermore, our physical model naturally enforces the memory capacity constraint of the mathematical formulation described in Equation 5.11. The hydrostatic potential energy term, $E_{hydrostatic_m}$, increases sharply as a node's available memory approaches zero. This effectively creates an infinite energy barrier that prevents placements violating the node's memory limits, thus ensuring the constraint is met.

7 | Ermes Framework Implementation

While the primary contribution of this thesis lies in the data placement strategies, and a comprehensive description of the implementation is provided in a companion work [44], this chapter offers an overview of the key components relevant to the performance evaluation. Accordingly, we focus our discussion on the implementation of the distributed data placement policy and the architecture of the *Ermes Node*, specifically its *Data Manager* modules, while providing a concise summary of the technologies utilized in the *Group Token Provider* (GTP) and *Function Registry* (EFR).

We begin by describing the implementation of the distributed data placement policy, introduced in Chapter 6, within the Ermes framework. Subsequently, we detail the foundational technologies of the system, analyzing their usage across the various components. We discuss the rationale behind each design choice, laying the groundwork for the experimental evaluation presented in the next chapter, where we examine their impact on overall framework performance. Finally, we focus on the *Ermes Node*, providing a detailed description of the components that support the implementation of the distributed data placement policy.

7.1. Implementation of the Distributed Data Placement Strategy

Chapter 6 describes the distributed data placement mechanism from a theoretical perspective. We detail how distinct consistency guarantees, and specifically their divergent replication behaviors, motivate the design of a dual-policy distributed protocol. This approach tailors the management strategy to the specific constraints of each consistency model supported by our framework. For eventually consistent collections, we employ an "always-copy" policy based on a distributed caching mechanism presented in Section 6.1. Since update propagation latency is independent of the replica count in this model, the

system proactively instantiates a new replica on a node closer to the user immediately following a remote function execution. In contrast, the policy governing sequentially consistent collections is significantly more complex. It strictly adheres to the energy minimization objectives defined in the physical model (Section 6.2), following the derived decision rules to determine whether to migrate the leader (Section 6.3.2) or create a new replica (Section 6.3.3).

This section bridges the gap between the theoretical formulation of the data placement strategy and its concrete implementation within the Ermes framework. We begin by detailing the local state information required by each node to enable placement decisions and support the data discovery phase, detailed in Section 4.5.3, in Section 7.1.1. Next, we present the dynamic extension of the physical model, transforming decision rules derived from an instantaneous energy minimization principle into a concrete distributed protocol capable of adapting to evolving network and workload conditions over time (Section 7.1.2). Finally, since the decision rules derived from the two policies govern the replication or migration of a collection to an adjacent node, they do not explicitly define criteria for data retention or offloading to higher tiers, we describe the eviction strategy in Section 7.1.3, which is common to both data placement policies.

7.1.1. Limited Knowledge on System State

To enable *Ermes Nodes* to make independent decisions regarding data placement and function routing during the data discovery phase, we require each node to maintain specific local state information. This state encompasses collection placement, resource utilization metrics (e.g. memory pressure), and data access patterns. We detail these components in the following paragraphs.

Collection Placement

In Ermes, each node maintains knowledge of all collections located locally or within the subtrees rooted at its children, including their specific consistency levels. We use this information to locate collections during the data discovery phase, along the network hierarchy, and to prevent redundant replica transmission to subtrees that already host the data. Specifically, a node knows whether a collection is reachable via one of its descending links, without necessarily knowing if it resides directly on the adjacent child or on a descendant node further down the hierarchy. We model collection placement information within the Ermes node using a state pattern. A known collection exists in one of two states: *local*, if the collection is stored on the node itself, or *remote*, if it is stored within a descendant's subtree but not locally. If a node has no knowledge of a collection, it

assumes by default that the data resides at a higher tier. In both states, the node tracks replicas located along descending paths. For sequentially consistent collections, we store additional information regarding the leader's location, which, similar to the collection itself, can be local, remote (reachable via a specific descending link), or unknown (assumed to be upstream). To maintain consistency in location data, each node propagates information about the creation of new collections to its ancestor. Furthermore, when a node migrates or replicates a collection to a descendant, it updates its local routing table to reflect the presence of the data along the corresponding descending link. Conversely, when offloading a collection to a higher-tier node, the offloading node informs its ancestor whether a replica remains within its subtree. If no replica remains, the ancestor removes the corresponding routing entry associated with that descending link.

Network State

Ermes nodes maintain information regarding the memory usage of adjacent nodes and the transmission latency of the corresponding links. We exchange this data periodically between nodes, as it is required to compute the decision policy for sequentially consistent collections derived from the physical model. We also leverage memory usage data to prevent the "always-copy" policy from overloading nodes with eventually consistent collections. Specifically, we suspend replication toward a target node until updated information is received if either of two conditions is met: (1) the reported memory usage exceeds a predefined threshold; or (2) a replication attempt is rejected by the destination node because its Least Recently Used (LRU) cache is saturated with active collections that cannot be evicted.

Access Patterns

Ermes nodes maintain usage statistics exclusively for locally stored, sequentially consistent collections, as these metrics are required solely for the energy minimization-based decision policy. Specifically, we track usage counts, which are incremented whenever the node locally executes a function accessing a specific collection. Furthermore, we monitor the cumulative access latency, defined as the sum of the full-path latencies (measured from the client entry point to the execution node) for all invocations targeting the collection.

7.1.2. Dynamic Extension of the Physical Model

While the physical model presented in Section 6.2 computes an optimal placement for a static system state, real-world operation requires adaptation to temporal variations in access patterns and resource availability. To address this, we designed a distributed

protocol that executes periodically to keep the knowledge of the system state at the *Ermes Node* updated. Although collection placement and data access patterns are updated in real-time, we restrict the exchange of network state information to specific intervals to minimize network overhead. Each node operates according to a three-state protocol whose states are:

- **Idle:** In this state, the node gathers real-time statistics on access requests that pass through it.
- **Gossip:** In this state, the node propagates its local state, including current memory usage, to its adjacent nodes in the hierarchy. During each gossip phase, a node sends exactly one message to each neighbor and waits until it has received a message from each of them.
- **Decision:** In this state, the node re-evaluates the placement of its managed collections. By applying the energy minimization formulas from our physical model to compute the new optimal locations for the leader and follower replicas, the node can initiate any necessary migrations.

Figure 7.1 shows a schematic representation of the three-state protocol illustrating the state transitions of the *Ermes Node*. In particular, how the node transitions between the *Idle* and *Gossip* states in response to either timer expiration or the receipt of a gossip message from an adjacent node. This mechanism induces a form of partial synchronization among neighboring nodes within the network. The transition from the *Gossip* state to the *Decision* state occurs once the node has acquired all the state information required to execute the placement logic.

We implement the *Decision State* using a concurrency pattern that evaluates placement decisions in parallel across distinct local collections, thereby avoiding synchronization bottlenecks. As depicted in the lower part of the figure, multiple routines share a common communication channel. This architecture comprises a single consumer, responsible for collecting outcomes, and multiple decision-makers. The latter evaluate placement policies by applying the decision rules for leader placement (Section 6.7) and the rule derived from the physical model for follower replicas (Section 6.9). These decision-makers publish their results to the shared channel; subsequently, the consumer aggregates the outcomes and executes the corresponding actions once all collections have been processed. Upon completion of these updates, the protocol reverts to the *Idle* state.

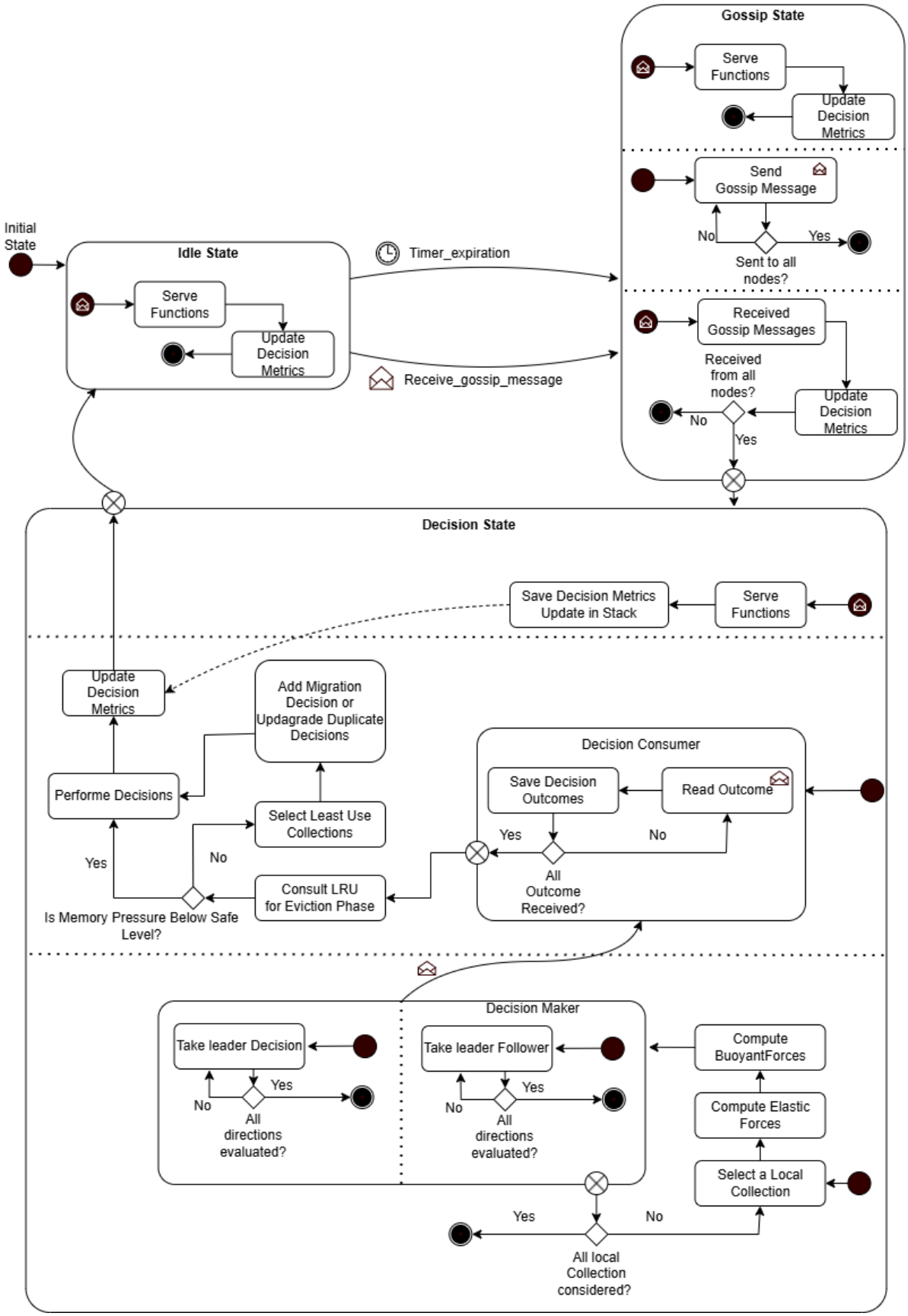


Figure 7.1: Three-state protocol description.

A complete cycle of these three states constitutes a single time window or *Epoch*. Metrics such as request counts and cumulative latency are aggregated within each window. To base decisions on longer-term trends rather than short-term fluctuations, our protocol can be configured to consider a history of multiple windows. This mechanism includes a configurable weighting function to reduce the influence of data from older windows, thereby prioritizing more recent access patterns.

For sequentially consistent collections, placement and migration decisions generally take precedence over function execution. However, to avoid the resource waste associated with preempting active functions, we apply these decisions based on their compatibility with currently running processes. Operations that strictly involve the creation of new read-only replicas, whether as a standalone duplication or the provisioning phase of a migration, do not conflict with active serverless functions and are executed immediately. In contrast, operations that modify the authoritative state or remove local data utilize a mechanism we term *Postponed Decisions*. Specifically, while the destination replica is instantiated immediately during a migration, the deletion of the local source copy is deferred. Similarly, decisions regarding Leader replica placement are postponed in their entirety. These deferred actions are queued and subsequently processed once the conflicting function terminates. To guarantee consistency during this interim period, any new requests targeting a collection with pending *Postponed Decisions* are refused.

7.1.3. Eviction strategy

The primary objective of the Ermes framework is to minimize average user-perceived latency. To achieve this, our data retention and offloading policies rely on the principle of temporal locality, assuming that recently accessed data is likely to be requested again in the near future. Consequently, we retain collections on a node until resource constraints necessitate their removal. We track collection usage via a Least Recently Used (LRU) mechanism. Upon receiving a new collection, if the node lacks sufficient memory, we offload the least recently used collections to a higher tier to reclaim local storage space. Additionally, Ermes supports a configurable memory safety threshold. During each *Decision Phase*, the node verifies whether its memory usage remains below this limit; if the threshold is exceeded, we preemptively release memory by offloading collections according to the same LRU policy.

7.2. Technology Stack of the Ermes Framework

We developed the Ermes platform following a modular design principle, selecting specific technologies tailored to the architectural role of each component. Figure 7.2 illustrates the technology stack for the three main components of the Ermes framework (section 4.2): the *Ermes Node*, the *Group Token Provider* (GTP), and the *Function Registry* (EFR).

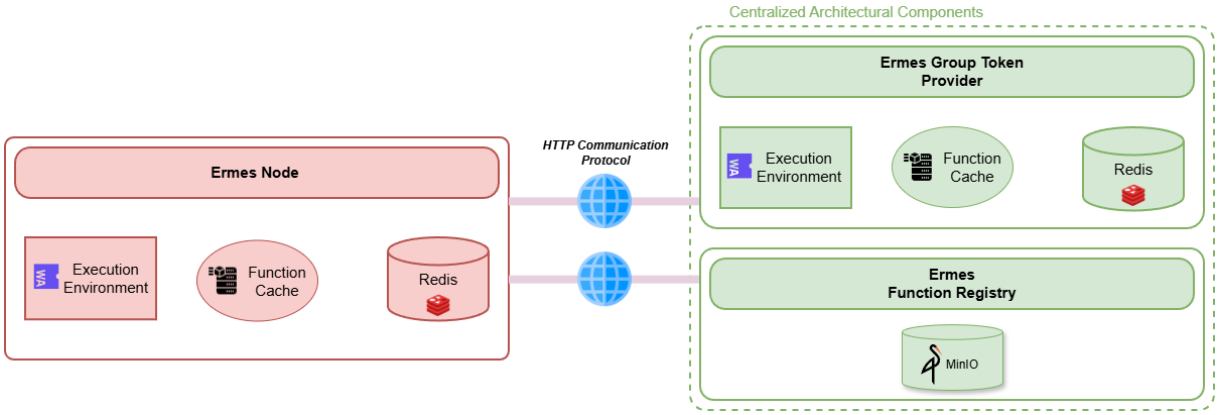


Figure 7.2: Ermes Technology Stack.

We implemented the core components of the Ermes framework in Go [21], a choice driven by its suitability for stateful, distributed systems. Go's concurrency model, based on lightweight *goroutines* [23] and channels, facilitates scalable parallelism and synchronization. Furthermore, the runtime provides automatic garbage collection, ensuring memory safety with minimal performance overhead.

Given the distributed nature of Ermes, efficient and secure inter-component communication is paramount. To ensure consistency and maintainability, we adopted a unified strategy for the network layer. We implemented HTTP interfaces using Go's standard library, avoiding the overhead and dependencies associated with heavyweight third-party frameworks. This approach leverages the library's robust networking and I/O capabilities, providing low-level control over request handling and ensuring high performance for communication-intensive tasks. On top of this HTTP foundation, we defined a custom communication protocol to facilitate state management, node interoperation, and routing. HTTP endpoints are classified as either public or private; private endpoints are accessible only to other *Ermes Nodes* presenting a shared secret token. This implementation inherently handles high concurrency, as the Go `http` server automatically spawns a new goroutine for each incoming request.

Both *Ermes Nodes* and the GTP support the execution of serverless functions: the former

handle user-invoked workloads, while the latter execute developer-defined authentication logic. To support these capabilities, our platform requires a runtime that accommodates multiple source languages while guaranteeing security, efficiency, and strong isolation. To address these goals, we selected WebAssembly (WASM) [25] as the core technology for our function execution engine. WASM provides a lightweight, sandboxed, and platform-independent environment that aligns with the requirements of a geo-distributed serverless architecture. By enforcing memory safety and restricting system calls, the runtime prevents functions from interfering with the host system or concurrent executions. Furthermore, the compact and portable nature of WASM binaries facilitates efficient distribution and deployment across our network of nodes.

While WebAssembly provides a standardized bytecode and execution model, distinct runtimes exhibit varying trade-offs regarding raw performance, initialization latency, and resource consumption. To maximize flexibility and enable environment-specific tuning, we designed Ermes with a pluggable runtime architecture. Despite deferring a detailed comparison of available runtimes to our companion work [44], we note that all performance evaluations presented in this thesis utilize *Wasmtime*, a leading runtime recognized for its high performance and strict adherence to standards.

Although WebAssembly runtimes are generally lightweight, minimizing per-invocation latency remains critical in high-throughput Edge scenarios. The overhead associated with creating a new runtime environment, configuring it, and linking the required host functions can introduce non-negligible latency. To mitigate this overhead, particularly in high-performance contexts, Ermes implements a pluggable *ExecutorPool* abstraction. This pool adapts its behavior to the specific capabilities of the selected runtime. For runtimes designed for high concurrency, such as *Wasmtime*, the pool leverages shared, thread-safe objects. Specifically, a single, heavy-weight Engine (in Wasmtime) or Runtime (in Wazero) is shared across all executions. When processing a request, the system acquires a worker slot and creates a new, lightweight execution context (e.g. a Store or Instance) for that specific call. This model ensures isolation by default while amortizing initialization costs, as only the ephemeral execution state is created and destroyed. Similarly, we pre-link the Ermes Host Functions API to shared objects during pool initialization, thereby amortizing the cost of host function registration across numerous invocations.

Although WebAssembly instances are inherently stateless, our framework supports stateful computation by granting seamless access to persistent data collections. As detailed in Section 4.4, we persist *function state* within a storage layer organized into collections, which are co-located on the execution nodes. We implement this layer using Redis [55], a NoSQL key-value store. This design choice addresses the limitations of traditional

relational databases, which enforce ACID semantics and consequently face scalability and availability constraints in wide-area distributed environments. Conversely, NoSQL systems are designed to manage the trade-offs of the CAP theorem, typically prioritizing availability and partition tolerance [17]. Consequently, they are well-suited for latency-sensitive and elastically scalable workloads. Moreover, as discussed in [15], hybrid key-value stores naturally support horizontal scalability, efficient partitioning, and high availability. These properties align with the requirements of distributed architectures demanding rapid access to co-located state.

User-invoked functions interact with the underlying Redis instance through *Host Functions*. These functions are implemented on the *Ermes Node* (host) and exposed to the WebAssembly instance (guest), allowing the sandboxed code to invoke external operations. This mechanism enables us to define a secure, custom API for state management.

In contrast, the EFR does not execute serverless functions but requires the persistence of large unstructured data, specifically WASM binaries and *Query Templates*. To decouple the storage layer from the application logic, this component employs MinIO [1], a high-performance object storage system. Instead of relying on complex hierarchical directory structures, MinIO utilizes a flat address space where binaries are stored as objects paired with rich custom metadata. This approach simplifies data retrieval and management.

To further optimize execution performance and minimize end-to-end latency, specifically by reducing the overhead associated with repeated network retrieval and runtime initialization, Ermes employs a *multi-layer caching mechanism* on nodes capable of executing serverless functions. This strategy comprises two logical layers: *Layer 1*, which targets the reduction of network and I/O latency by caching function artifacts; and *Layer 2*, which optimizes CPU usage and startup times by managing compiled executable modules in memory, effectively bypassing the compilation step for locally available binaries.

7.3. Stateful Capabilities in the Ermes Node

Having described the foundational technologies utilized throughout the Ermes framework, we now focus on the *Ermes Node*, paying particular attention to the components that enable stateful execution, namely, the distributed data placement policy and the function scheduling procedure. The *Ermes Node* serves as the fundamental computational and storage unit in our system, responsible for executing functions and managing state across the distributed network. We deploy these nodes within a hierarchical topology, where

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

nodes located at higher tiers typically offer greater computational resources.

A critical design feature is adaptability. Our nodes support the dynamic reallocation of data collections across the network, a mechanism that allows the system to balance load and move data closer to clients. This characteristic is essential for maintaining high performance and availability in heterogeneous and evolving environments.

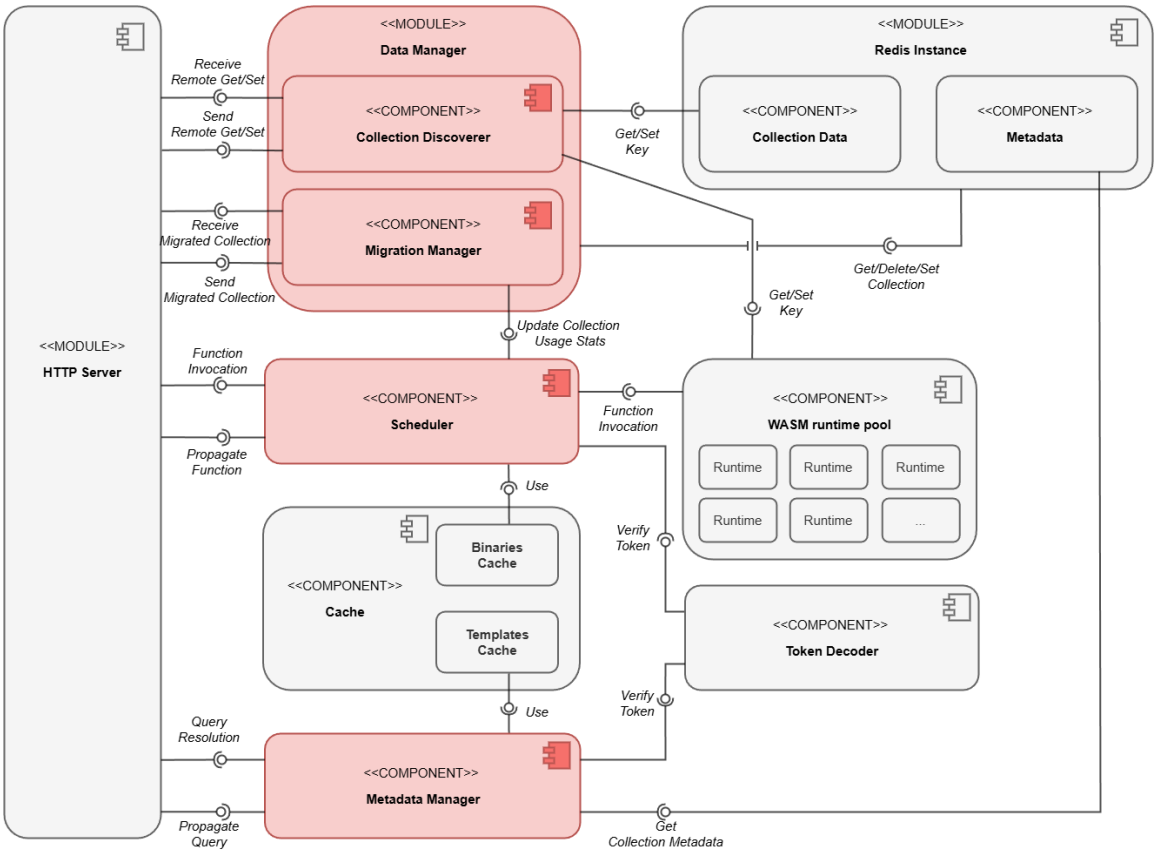


Figure 7.3: Ermes Node Architecture.

Internally, we designed the *Ermes Node* with a modular architecture that separates core concerns. Distinct software components handle state management, function execution, and data coordination. This separation enhances maintainability and extensibility, allowing us to modify or replace individual mechanisms, such as replication policies or execution strategies, without altering the node's external interfaces. Figure 7.3 provides a high-level schematic of these internal components. The colored elements indicate the components relevant to this thesis, which we detail in the subsequent sections. Conversely, the grey elements represent the supporting infrastructure, including: the *HTTP Server*, which acts as the entry point to the *Ermes Node* for both client requests and inter-node communication; the multi-level cache system; the *Underlying Storage*, implemented using Redis;

the *Execution Environment*, managing the WASM runtime pool; and the *Token Decoder*, responsible for validating signed tokens to enforce the RBAC mechanism.

7.3.1. Metadata Manager

The *Metadata Manager* is the component responsible for managing the metadata lifecycle. Its primary function is to process and resolve *Metadata Queries* during function invocation to identify the required data collections. This step is fundamental for determining the set of collections accessed by a function prior to its execution, thereby enabling the *Data Discovery* phase to optimize function placement within the Ermes hierarchical network. Additionally, we utilize this information to update the decision metrics required during the *Decision* phase for sequentially consistent collections, specifically when evaluating placement rules.

In our design, collection metadata is anchored to a fixed, predetermined geolocation, whereas the actual data can migrate freely across the infrastructure. Consequently, a *Metadata Query* must be executed on the specific *Ermes Node* responsible for the corresponding location. The *Metadata Manager* on a given node is therefore responsible for resolving queries that fall within its domain and forwarding all other requests to the appropriate node in the hierarchy.

Initially, the EFR stores all function metadata and query templates, while we pre-position collection metadata on the responsible *Ermes Nodes*. When a function is invoked for the first time on a node, its *Metadata Manager* fetches the relevant function metadata and associated query templates from the EFR. Upon resolving the query, the manager caches these artifacts locally to accelerate subsequent lookups.

7.3.2. Scheduler

The *Scheduler* serves as a core component of the *Ermes Node*, responsible for orchestrating the execution of Application Functions. It manages the injection of the execution context into the WASM instance and implements the routing logic detailed in Section 4.5.3. Our implementation translates these high-level policies into a concurrent, high-performance execution engine. Leveraging the node's *Limited Knowledge of Collection Placement*, the *Scheduler* determines whether to execute the function locally or offload it to an adjacent node. When the scheduling policy dictates remote execution, the *Scheduler* acts as an HTTP client, constructing a mirrored request and forwarding it to the target node. Our implementation of the scheduling policy incorporates several optimizations to enhance robustness and efficiency beyond the theoretical decision tree. Notably, when selecting an

offloading target among multiple valid candidates, the system utilizes neighbor latency metrics to deterministically select the physically closest node. It is important to note that this selection optimizes for the immediate next hop rather than the final destination; since nodes possess only aggregated information regarding their subtrees, they cannot distinguish whether the target collection resides directly on the adjacent node or deeper within the hierarchy.

The *Scheduler* is also responsible for handling access denials for sequentially consistent collections caused by out-of-order invocations or session guarantee violations. We address these scenarios through a retry mechanism. If a scheduled function requires a collection version newer than the one stored locally, access is denied. The node then initiates a bounded retry loop, waiting for the missing update. If the maximum number of retries is exceeded, the Application Function invocation fails. Additionally, we incorporate a loop detection mechanism to prevent infinite routing cycles within the hierarchy. This safeguard mitigates potential routing anomalies caused by transient unavailability during concurrent collection operations.

To optimize performance, the *Scheduler* manages persistent updates asynchronously relative to the user function invocation. By spawning a separate Goroutine, we avoid imposing unnecessary latency on the user for database insertion and update propagation. The only exception to this approach concerns local write operations on Sequentially Consistent collections. To guarantee client-centric consistency, the system must return the new Sequence Number associated with the collection to the client. Consequently, the user incurs a minimal latency overhead to allow for the local update and the retrieval of the new Sequence Number within the HTTP response. However, the propagation of these updates to remote replicas remains asynchronous, ensuring that the user does not wait for global synchronization across the network.

Finally, to facilitate flexibility and simplify the evaluation phase, we adopted the Strategy Pattern, allowing the system to switch between different scheduling behaviors without modifying the core execution logic. This design supports a simplified scheduler variant used when Ermes is deployed with a centralized database. In this configuration, primarily used as a baseline for evaluation, functions execute locally while retrieving data remotely. A *Base Scheduler* component encapsulates the logic shared across these different strategies.

7.3.3. Data Manager

The *Data Manager* module enables stateful capabilities for serverless functions executing on *Ermes Nodes*. It comprises two primary components: the *Collection Discoverer*, which provides a unified data access abstraction to mask the complexity of local and remote operations, and the *Migration Manager*, which implements our distributed protocol for data placement and migration.

These components share and maintain the *Limited Knowledge of Collection Placement*, ensuring that placement decisions are informed by consistent state regarding local storage and subtree contents. To guarantee a unique instance within each *Ermes Node*, we implemented both components using the Singleton design pattern.

The *Collection Discoverer* fulfills two key roles. First, it assists the *Scheduler* by identifying locally stored collections. Second, it acts as middleware between the WASM runtime and the underlying Redis instance, managing all read and write operations on function state. By exposing methods that support single and multi-key operations, it abstracts the distinction between local and remote access. This abstraction is fundamental to our data placement and scheduling policies. While the physical model assumes that functions access a single collection locally, our implementation relaxes this constraint for eventually consistent collections, permitting remote access. Consequently, since the scheduling algorithm places functions on the first available node hosting a subset of the required data, it is crucial to support remote access transparently without burdening the developer with additional complexity.

We perform read operations synchronously; while remote reads incur network latency, their results are cached within the function-private scope. In contrast, we employ a write-back policy for write operations. Modifications are buffered locally during execution and committed to the key-value store, and propagated to remote replicas, in a single batch upon function termination. This approach significantly reduces network overhead and latency.

However, this design introduces a limitation regarding writes to eventually consistent collections. A write operation targeting a remote collection that is concurrently removed results in a silent failure, as the node cannot distinguish between an unknown collection and one that no longer exists. This issue does not affect sequentially consistent collections, as their write operations are invariably routed to the leader replica and executed locally.

We route remote reads and update propagations through the hierarchical network. Read requests follow a path similar to function scheduling: they move downstream if the node is

aware of the collection in a child's subtree, and upstream otherwise. Conversely, update propagations must reach all replicas and are therefore always forwarded upstream to ensure broad dissemination.

The *Collection Discoverer* is also responsible for enforcing consistency. It resolves conflicts on eventually consistent collections using a Last-Writer-Wins (LWW) policy, discarding incoming updates with older timestamps. Furthermore, it manages the client session guarantees, implemented as vector clocks [52] for each collection. If vector clock verification for a sequentially consistent collection fails, the system denies access and aborts the function execution.

The *Migration Manager* is responsible for maintaining the *Limited Knowledge of Collection Placement*, dynamically tracking creations, deletions, and migrations. This component executes the migration policies and the state protocol detailed in Section 7.1. We utilize the State design pattern to implement the policy for sequentially consistent collections, mirroring the underlying three-state protocol. To inform placement decisions, the *Migration Manager* aggregates access metrics, such as request frequency and latency, within a circular buffer. Each slot represents a specific *Epoch*, with the buffer's capacity defining the total temporal window. To prioritize recent system states, we apply a configurable weighting function to these historical data points; currently, we support logarithmic and polynomial decay functions. Finally, to enhance maintainability and support multiple data migration policies, we implement this component using the Factory design pattern.

7.4. Ensuring Consistency in a Dynamic Environment

Eventual consistency guarantees that all updates will eventually propagate and be applied to every node in the system. Strong eventual consistency (SEC) further ensures that replicas that have received the same set of updates will converge to an identical state. Sequential consistency provides a stronger guarantee: it requires that all nodes observe the same global sequence of updates, consistent with the program order of each process, regardless of real-time execution order.

Both models rely on the assumption that replicas will converge to a coherent state once updates cease. However, without specific precautions, our architecture is susceptible to a rare condition that can lead to update loss and permanent divergence. This occurs when an update message and a collection replica traverse the same communication channel in opposite directions simultaneously.

To resolve this issue, the *Collection Discoverer* and the *Migration Manager* implement a temporary caching strategy that persists for the duration of the corresponding HTTP request. Specifically, during update propagation, the *Collection Discoverer* buffers updates intended for non-local collections. If the collection is received on that node while the update message is being processed, the system applies the update locally. Dually, the *Migration Manager* tracks the direction of active migrations; if an update message arrives and must be forwarded in the same direction as a migration in progress, we delay the forwarding until the migration has completed, ensuring the update reaches the intended replica.

7.5. Synthesizing the Implementation: Component Collaboration

In this chapter, we detailed the implementation of the Ermes framework, realizing the architectural design as a functional system. We discussed the rationale behind the selected technology stack, which ensures the performance, security, and concurrency required for a distributed serverless platform.

Our implementation achieves a strict separation of concerns. Two centralized components, the **Group Token Provider (GTP)** and the **Ermes Function Registry (EFR)**, serve as authoritative sources for system-wide state, managing access control policies and the function catalog, respectively. These components constitute the stable infrastructure supporting the distributed layer, which is structured as a hierarchy of nodes. At the core of the system lies the **Ermes Node**, a modular entity that integrates a secure **Execution Environment** for WASM serverless functions. Crucially, the node incorporates the **Data Manager**, a component that implements the distributed data placement protocol, a primary contribution of this work, by dynamically migrating data in response to real-time access patterns.

The collaboration between centralized and distributed components enables Ermes to balance global consistency with local autonomy, facilitating data-aware function scheduling and adaptive resource management. Having detailed the system implementation, we proceed to the evaluation phase. In the subsequent chapter, we first validate the theoretical mathematical model to establish a rigorous foundation, and then evaluate the empirical performance of the implementation to confirm the effectiveness of our design choices.

8 | Ermes Framework Evaluation

This chapter presents the empirical evaluation of the Integer Linear Programming (ILP) model and the Ermes framework. We focus on validating the core architectural principles, with particular emphasis on the **data migration logic** inherent in our distributed data placement solution and the specific impact of the **stateful distributed architecture**. We refer to the companion thesis for a detailed discussion of implementation specifics and engineering-related performance metrics.

We first evaluate the centralized mathematical model in a simulated environment to assess its scalability, aiming to demonstrate the limitations that motivate the adoption of a distributed policy. Subsequently, we evaluate the Ermes framework to validate the architectural design and performance characteristics of our proposed system. Our primary objective is to assess the system against its key requirements, including user-perceived latency, responsiveness under dynamic workloads, and the maintenance of data correctness and consistency.

Specifically, the evaluation seeks to validate whether the observed data movement aligns with our theoretical models, ensuring that the system behaves predictably under diverse workloads. Furthermore, we aim to isolate and quantify the contributions of individual architectural features to understand how specific design choices influence the system's overall responsiveness and scalability.

8.1. Research Questions

Our experimental campaign is structured to empirically address six core Research Questions (RQs), which collectively serve to validate the architectural principles and performance characteristics of the Ermes framework:

RQ1 To what extent do system dimensions (nodes, collections, and functions) and consistency requirements dictate the asymptotic scalability of centralized data placement optimization, and at what threshold does the latency of finding an optimal solution compromise its feasibility for real-time orchestration in dynamic

edge-cloud environments?

RQ2 Does the runtime behavior of our data migration policy align with the theoretical predictions of the mathematical model, and does the system reliably converge to the expected equilibrium states?

RQ3 How do different consistency models impact overall system performance, and how do they influence the migration behavior of the data placement policy?

RQ4 How does our stateful, distributed architecture compare against traditional centralized and stateless FaaS paradigms in terms of key performance metrics?

RQ5 What are the individual performance contributions of key architectural mechanisms, such as data locality and client-side caching?

RQ6 How robust and scalable is our framework when subjected to dynamic operational conditions, including stochastic access patterns and user mobility?

To answer these questions, we analyze the system's **temporal evolution** and its **steady-state performance**. To address **RQ2**, we first characterize the system's behavior during its **transient phase**, measuring the **convergence time** required to reach a stable equilibrium. Crucially, we verify that the final data placement observed in our implementation matches the **optimal placement** predicted by our theoretical model. Once this alignment is validated, we evaluate steady-state performance, which we define as a composite measure of two primary metrics: **user-perceived latency** and **network overhead**. User-perceived latency is the end-to-end response time experienced by the client, while network overhead quantifies the control and data traffic generated by our placement and consistency protocols.

8.2. Experimental Methodology

To address the research questions outlined above, which range from determining the scalability limits of centralized optimization to validating the performance and robustness of our distributed stateful architecture, we designed a comprehensive experimental campaign. Our methodology aims to empirically verify the alignment between the system's runtime behavior and theoretical predictions, while quantifying key metrics such as user-perceived latency and network overhead under diverse operational conditions. To ensure clarity and reproducibility in this assessment, we structure the evaluation into three complementary tracks:

1. **Mathematical Model Scalability**, which is designed to evaluate the scalabil-

ity properties of the centralized solution for data placement. We first analyze the asymptotic behavior of each variable in the system, then we scale the problem under high workloads and measure the execution time in order to provide empirical evidence to address **RQ1**.

2. **Use-case-driven experiments**, which are designed to evaluate our data migration policy. These tests verify that the system's runtime behavior converges to the placements predicted by our theoretical model and allow us to analyze the system's temporal evolution as the policy is applied. These experiments provide the empirical evidence needed to answer **RQ2** and **RQ3**.

3. **Ablation studies under stochastic access patterns**, which are aimed at quantifying the performance impact of specific architectural features. We first establish a baseline under realistic, stochastic workloads. We then systematically disable selected components to measure their individual contributions to overall performance, a process that allows us to address **RQ4**, **RQ5**, and **RQ6**.

8.3. Mathematical Model Scalability

The mathematical model introduced in Chapter 5 serves as a theoretical optimal baseline. In this section, we do not evaluate it as a production-ready component, but rather as a benchmark to explore the fundamental scalability limits of centralized optimization in serverless and edge environments, where workloads are bursty and can change abruptly. By analyzing its execution time, we aim to quantify the point at which the pursuit of absolute optimality becomes computationally intractable, thereby justifying the transition toward the decentralized heuristics proposed in Section 6.2.

This section experimentally explores these scalability properties. We first identify the benefits of the variable reduction discussed in Section 5.7 for the baseline model without replication (Section 5.3). Subsequently, we analyze how individual components affect model construction and solution time using the replication-aware models: eventual consistency (Section 5.4) and sequential consistency (Section 5.5). Finally, we investigate the limitations of model execution time by considering varying function and collection workloads.

8.3.1. Experimental Setup

We implemented all mathematical models using the *Concert Technology* of *IBM ILOG CPLEX*[1] in C++. Following the general formulation of the replication model in Equation 5.18, we frame the problem as a **multi-objective optimization**[2] employing *lexicographical ordering*. We first minimize function execution times, represented by the term $\mathcal{F}$, and subsequently minimize the regularization term $\mathcal{C}$ among the set of optimal solutions.

Our approach subdivides the optimization process into a construction phase and a solution phase. During construction, we define the decision variables, formulate the objective function, and specify the constraints. In the solution phase, we employ the CPLEX solver to compute a feasible solution. We include the internal presolve stage of the solver within the recorded solution time to capture the full computational effort required by the engine.

Certain constraints within the models require a sufficiently large constant $\mathcal{M}$ for feasibility. In particular, constraint 5.22 involves the sum of all collections accessed by function f, $k \in K_f^E$; thus, we set $\mathcal{M} = |K| + 1$ to ensure it is satisfied $\forall f \in F$.

To characterize how individual parameters influence scalability, we varied one primary variables (number of nodes $|N|$, number of collections $|K|$, or number of functions $|F|$) while holding others constant. To ensure a diverse range of network conditions, we synthetically generated the system state. The network is modeled as a complete graph where the latency and bandwidth for each edge (i, j) are sampled from continuous normal distributions: $L_{i,j} \sim \mathcal{N}(10, 2)$ ms and $B_{i,j} \sim \mathcal{N}(100, 20)$ Mbps, representing a heterogeneous edge-cloud environment. While these distributions determine the specific values of the coefficients in the objective function, previous work [40] has shown that the solver's computational complexity is primarily driven by the problem's dimensionality (the number of variables and constraints) rather than the specific numerical distribution of the input data. Finally, the read-write pattern was chosen to be 80% reads and 20% writes.

Each data point reported in our results represents the average of 10 independent trials using different random seeds to account for variance in the synthetic generation process.

We conducted each test on a server machine equipped with 16 AMD64 cores (AMD Ryzen 9 9950X, 32 vCPUs), 64GB of DDR5 RAM, and running Fedora Server 42. The C++ source code was compiled with `-O3` and `-march=native` flags to enable maximum compiler optimization.

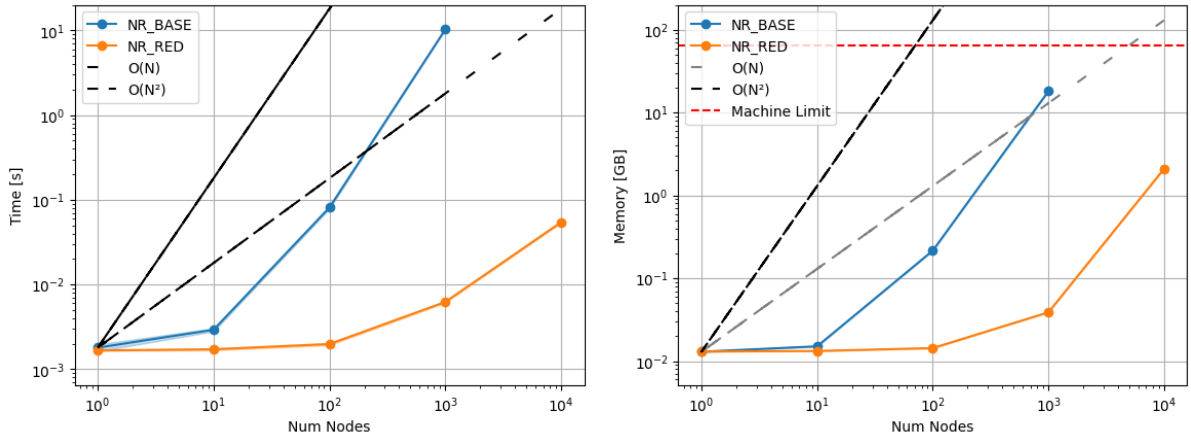
Since the CPLEX solver automatically selects the optimal branch-and-cut algorithm for

[1] https://www.ibm.com/products/ilog-cplex-optimization-studio
[2] https://www.ibm.com/docs/en/icos/22.1.1?topic=optimization-multiobjective

each instance[3], our analysis focuses on observed execution trends during the construction phase rather than the theoretical complexity of the underlying solver algorithms. By utilizing a high-performance solver and an optimized C++ implementation, we establish a performance upper bound for centralized orchestration. This setup allows us to measure the optimality gap and determine if a centralized approach can realistically accommodate the high-frequency state changes inherent in distributed edge-cloud systems.

8.3.2. Effectiveness of Variable Reduction

In this experiment, we varied the total number of nodes while fixing the number of functions and collections to 10. Figure 8.1 presents the mean execution time and memory consumption for both the baseline and reduced no-replication models. The results are plotted on a logarithmic scale, including reference lines for $O(n)$ and $O(n^2)$ scaling.



(a) Solving time of the baseline and reduced problems.

(b) Max Resident-Set-Size (RSS) of the baseline and reduced problems.

Figure 8.1: Effect of the variable reduction on the time and memory occupied by the solver.

The reduction in the number of variables lowers execution time (Figure 8.1a) and memory footprint (Figure 8.1b) by at least two orders of magnitude. Specifically, the baseline model can only accommodate problems with up to 1000 nodes due to its quadratic scaling in the number of variables, as discussed in Section 5.7.1.

A clear performance gap exists between the baseline and reduced models, with the latter maintaining a sub-second execution time for instances up to $N = 10000$. Although the number of variables becomes linear, as reflected in the asymptotic execution time, memory

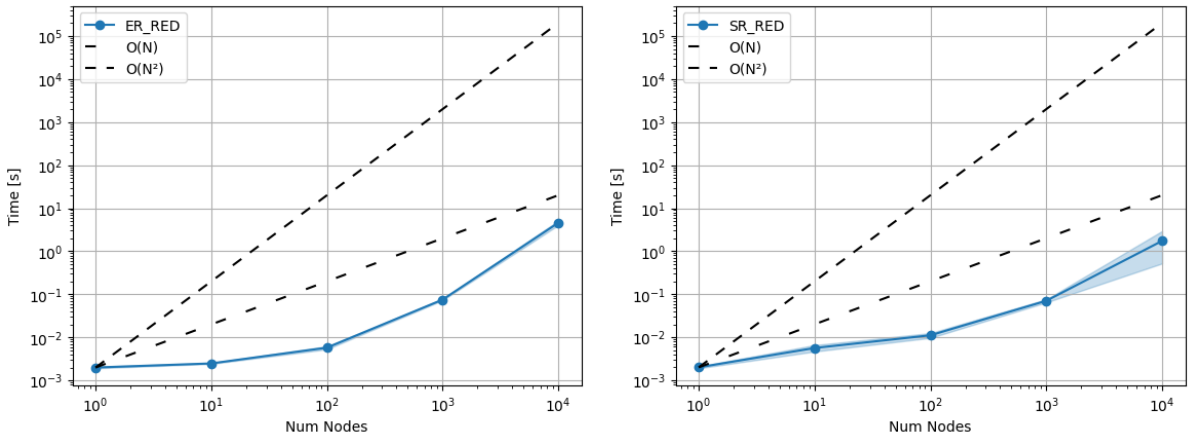
[3]https://www.ibm.com/docs/en/icos/22.1.1?topic=c-ilocplex

scaling for large problems remains superlinear. This behavior is expected because model complexity depends on the variables, constraints, and the dimensions of the objective function. In this case, both the objective function size and the number of constraints, such as the collection uniqueness constraint in Equation 5.13, scale with the number of nodes.

8.3.3. Node-Count Scalability

To evaluate model properties independently, we use the reduced versions of the sequential and eventual consistency replication models as references. These models provide deeper insights into scaling properties and allow for testing larger instances. For this section, we varied the number of nodes from 1 to 1000 while fixing the collections and functions to 10.

Figure 8.2 presents the scalability plots for both consistency models on a logarithmic scale.



(a) Eventual Consistency model solution time. (b) Sequential Consistency model solution time.

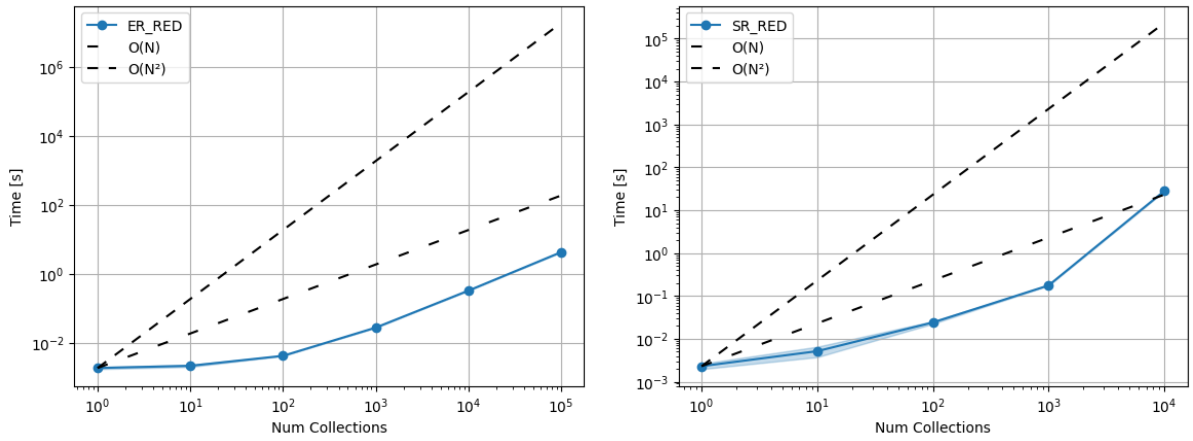
Figure 8.2: Scalability of the eventually and sequentially consistent models when varying the number of nodes.

The total execution time for the reduced sequential consistency model (Figure 8.2b) appears to scale linearly for large instances, as expected since the number of variables follows a linear trend (Section 5.7.2). Interestingly, for large networks, we encounter a larger variance in the solution time. This increase can be attributed to the strict dependency of the problem to initial conditions of the problem. The problem can be solved very quickly in some particular instances where functions are already placed in the same node as the collection it accesses.

Conversely, the eventual consistency model (Figure 8.2a) exhibits superlinear scaling for large problems. The the C++ construction logic includes a pre-processing step to calculate the closest replica for every possible node-destination pair, as described in Equation 5.16. Since this operation is dependent on the number of nodes, as it is performed for all node pairs (i, j) associated with the movement variable $z_{i,j}^{(f)}$, this trend is more visible with larger instances, leading for very large (but unrealistic) networks to a cubic scaling $O(N^3)$.

8.3.4. Collection-Count Scalability

In this experiment, we evaluated the reduced replication models under high and low replication densities while fixing the number of nodes and functions to 10. This configuration allows us to test instances with up to 100,000 collections. Figure 8.3 illustrates the scalability results on a logarithmic scale.



(a) Eventual Consistency model solution time. (b) Sequential Consistency model solution time.

Figure 8.3: Scalability of the eventually and sequentially consistent models when varying the number of collections.

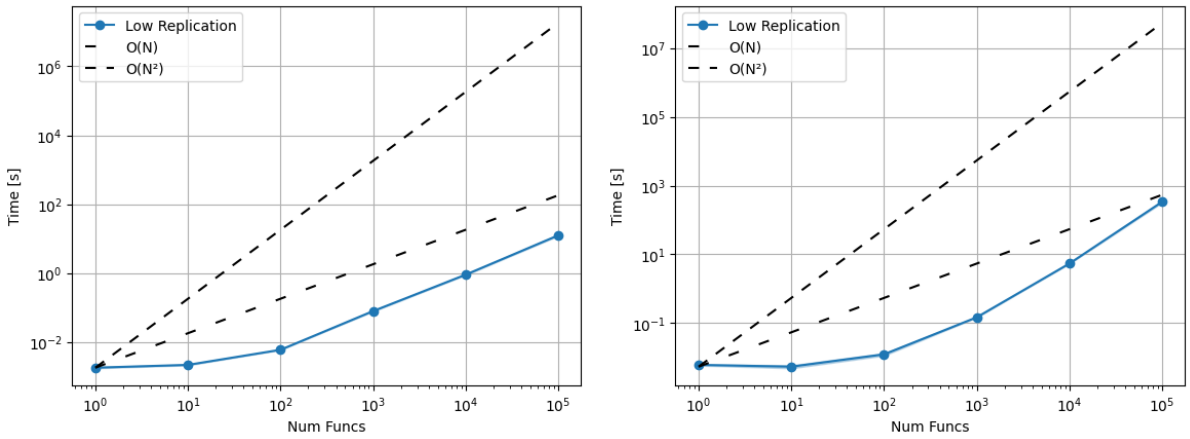
Both models exhibit linear scaling in execution time for low collection counts (Figures 8.3a and 8.3b). The eventual consistency model here does not display an asymptotic increase in the scaling because the fixed, small node count ensures that finding the closest replica node contributes only a constant offset. For both models, the linear scaling of variables, constraints, and objective function size is reflected in the solution time.

Notably, the sequential consistency model exhibits super-quadratic asymptotic complexity as the number of collections increases. At 100,000 collections, the solution time exceeds our one-hour threshold, suggesting non-polynomial scaling. This may be explained by

the regularization term $\mathcal{C}$, which, when normalized by l_{max}, produces numerous near-optimal solutions. Combined with strict function placement policies, this transforms the branch-and-cut algorithm into a highly complex combinatorial task, as the effect of the regularization term has minimal effect on the value of the objective function.

8.3.5. Function-Count Scalability

We examined the reduced replication models under both replication density scenarios, fixing the nodes and collections at 10 and testing up to 100,000 functions. Figure 8.4 shows the corresponding scalability plots.



(a) Eventual Consistency model solution time. (b) Sequential Consistency model solution time.

Figure 8.4: Scalability of the eventually and sequentially consistent models when varying the number of functions.

For small problem dimensions, up to 100 functions, scaling remains constant (under 100 ms); however, larger instances reveal two distinct trends. The eventual consistency model (Figure 8.4a) scales linearly, whereas the sequential consistency model (Figure 8.4b) exhibits quadratic scaling. This discrepancy stems from the tighter coupling between functions and accessed collections in the sequential model. With limited collections, each becomes increasingly constrained by multiple functions, particularly those requiring execution on a collection's leader node. While eventual consistency permits more flexibility in function placement, the sequential model must explore more branching paths, leading to increased solution time.

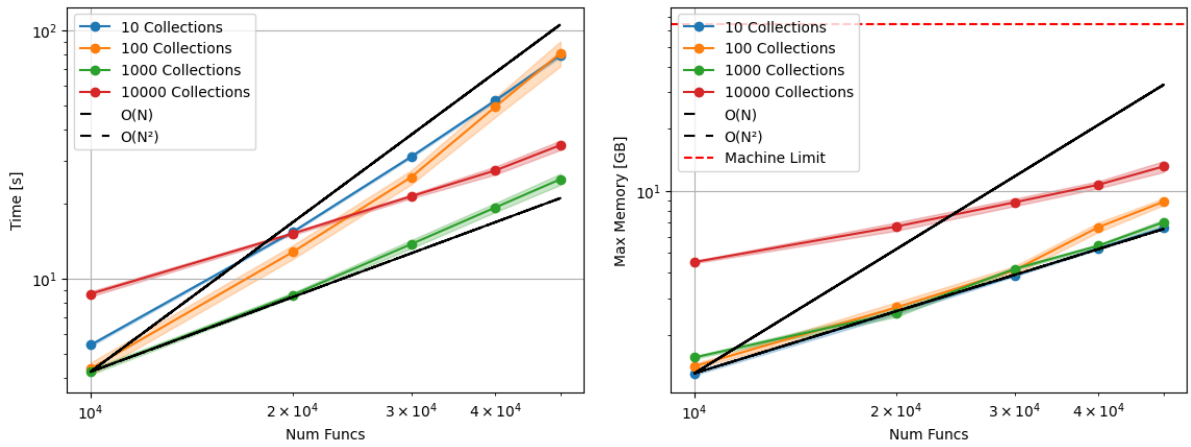
The observed inverse correlation between the number of collections and solution time can be attributed to **problem decoupling**. With fewer collections, functions are tightly coupled by shared data dependencies, creating a dense constraint matrix that forces the

solver to exhaustively explore a complex search space. Conversely, increasing the number of collections increases constraint sparsity; since functions interact with distinct data subsets, the solver can prune the search tree more efficiently, finding an optimal configuration with fewer combinatorial conflicts

8.3.6. Scalability Under High Workload

The preceding experiments confirm that the asymptotic execution times align with theoretical predictions and are heavily dependent on the number of functions, collections, and nodes. While we have analyzed isolated behaviors, we have also observed inherent coupling effects of complex optimization systems. To investigate these combined properties, we now analyze scenarios featuring both high function and collection counts.

Since node count is the primary driver of complexity, and real-world systems cannot scale nodes indefinitely, we fix the node count to $|N| = 10$ for this evaluation. We use the **reduced sequential consistency model with low replication density**, as it is the most computationally intensive. We performed 10 executions, varying $|K|$ from 10 to 10000 and starting from a baseline of 10,000 function invocations.



(a) Solution time of the problem. (b) Max Resident-Set-Size (RSS) of the problem.

Figure 8.5: Time and memory profiles of the model under varying number of functions and collections.

Figures 8.5a and 8.5b illustrate the timing and memory profiles. Interestingly, the results reveal an inverse correlation: increasing the number of collections for a fixed number of functions appears to decrease solution time asymptotically while memory consumption continues to increase linearly. This supports the conclusion that a higher collection count increases constraint coupling. When a small set of collections is accessed by a large

number of functions, the resulting representation matrix becomes significantly denser. In optimization terms, this high degree of interdependency limits the solver's ability to decompose the problem or effectively prune the search space, and forces the branch-and-cut algorithm to explore a much larger portion of the possibility tree before confirming optimality.

A clear performance gap is visible for instances with 10.000 collections, where the regularization term increases problem difficulty. As the number of functions increases, the set of initial solutions found in the first multi-objective step decreases, simplifying the resolution of the regularization term and explaining the observed timing and memory gaps.

While low workloads can yield optimal solutions within 10 seconds, higher workloads result in solution times between 20 and 80 seconds. This indicates that the model is **operationally incompatible** with edge environments where invocation patterns or topologies change at sub-second scales. Furthermore, we observed that increasing the network size to $|N| = 100$ results in a minimum solution time of 180 seconds and 20 GB of memory even for small workloads. Doubling the workload to 20.000 functions increases solution time to over 10 minutes and exhausts the 64 GB memory limit.

These results confirm the computational intractability of the centralized optimal model, especially for real-time edge to cloud orchestration. As shown in Figure 8.5a, the time required to find an optimal solution under high workloads exceeds the operational window of most edge applications. This state obsolescence, where the network state changes faster than the solver can find a solution, conclusively demonstrates that while the ILP model provides an invaluable theoretical benchmark, a practical framework must rely on the near-optimal, low-latency distributed protocols that we further analyze in the following sections.

8.4. Ermes Experimental Testbed

Having demonstrated the scalability limitations of the centralized model in the previous section, we now turn our attention to the evaluation of the Ermes framework and its distributed data placement solution. In this context, we regard the implementation detailed in Chapter 7 not as the primary subject of the evaluation, but rather as the scientific instrument used to validate our architectural hypotheses. This prototype provides the functional environment necessary to translate conceptual design choices into quantifiable metrics, such as latency, throughput, and resource utilization. Consequently, we structure the following experiments to systematically measure the performance implications of these design decisions, providing empirical evidence of their effectiveness in a practical,

distributed setting.

Figure 8.6 illustrates the schematic representation of the testbed network. To evaluate our system under realistic conditions, we deployed a virtualized testbed using Proxmox[4], designed to emulate a national-scale, multi-tier distributed network. This infrastructure comprises nodes characterized by distinct roles and heterogeneous resource allocations, as detailed below:

- **Centralized Services Node:** dedicate nodes (labeled 58-59 in Figure 8.6) hosts the GTP and EFR. They are configured with 8 CPU cores, 8 GB of RAM, and 30 GB of storage, with negligible network latency to all other nodes.
- **Cloud Node:** A single Tier 3 node (labeled 57) representing the IT top-level domain, configured with 8 CPU cores, 8 GB of RAM, and 40 GB of storage.
- **Fog Nodes:** Two intermediate Tier 2 nodes representing the IT/ROME (55) and IT/MILAN (56) domains, each configured with 4 CPU cores, 4 GB of RAM, and 20 GB of storage.
- **Edge Nodes:** Five peripheral Tier 1 nodes, two under the ROME domain (50 and 51) and three under the MILAN domain (52, 53, and 54). Each is configured with 2 CPU cores, 2 GB of RAM, and 10 GB of storage.

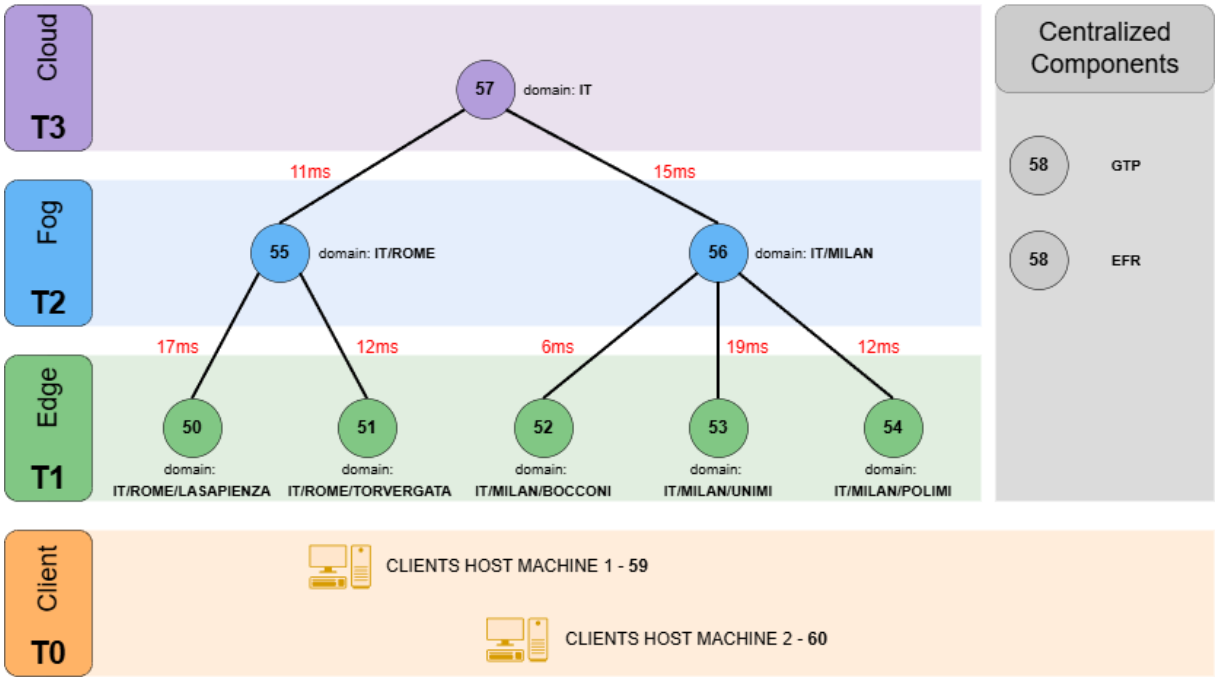


Figure 8.6: Ermes Testbed Network.

[4]https://www.proxmox.com/en/

We designed the Ermes testbed topology and resource allocation to mimic a real-world, geographically distributed deployment [42]. Our primary objective is to isolate the performance of distributed interactions within the network, with specific emphasis on the data migration logic. To this end, we configured negligible latencies for the links connecting centralized service nodes to the rest of the network, as well as for the connections between clients and their entry-point nodes. We adopt this approach because any fixed communication overhead associated with these endpoints would introduce a constant offset to our measurements without affecting the relative outcomes.

Furthermore, we employed fixed and controlled latencies between infrastructure nodes to ensure experimental reproducibility, enabling a systematic evaluation under predictable network conditions. Specifically, Figure 8.6 details the latencies between tiers, expressed as Round-Trip Times (RTT), which range from 5 ms to 20 ms. To avoid experimental bias, we intentionally avoided enforcing a uniform hierarchical latency pattern; for instance, the latency between Tier 2 and Tier 1 nodes is not strictly lower than that between Tier 3 and Tier 2 nodes. This configuration reflects the heterogeneous nature of real-world network links.

8.5. Use-case-driven experiments

The primary objective of this set of experiments is to assess the *transient behavior* and *convergence time* of the Ermes framework. We empirically verify the consistency between the behavior of our implemented data migration policies and the theoretical optimal placement derived from the mathematical model. To this end, we use the model solver, presented in the previous Section 8.3.1, to compute the optimal configuration for every collection involved in each experiment.

To ensure a rigorous evaluation, we ground our methodology in a representative use case that reflects a typical operational environment for the Ermes framework. Building on this foundation, we design a series of five deterministic scenarios.
As detailed in Section 4.5.3, when a function accesses a Sequential Consistency collection alongside one or more Eventually Consistent collections, the scheduling decision is primarily driven by the collection with the strictest consistency requirement, which necessitates local access. Furthermore, while the placement policy for Eventually Consistent collections follows a standard caching algorithm, the policy for Sequential Consistency is inherently more complex and warrants deeper investigation. Consequently, the first four experiments focus exclusively on the placement of Sequential Consistency collections, while the final experiment addresses a mixed-consistency scenario involving both consis-

tency guarantees. The sequence of experiments follows a path of increasing complexity. We first present simpler, conflict-free scenarios where a single collection is accessed by at most one writer and one reader. These scenarios represent standard patterns found in the literature, such as Top-Down Data Dissemination, Bottom-Up Data Aggregation, and Peer-to-Peer data sharing. Subsequently, the last two experiments introduce the additional complexity of conflicting access patterns, where multiple clients compete for the same collection.

Our data placement protocol for sequentially consistent collections, which implements the Physical Model, operates within discrete time windows designated as *Epochs*. An *Epoch* corresponds to one complete execution cycle of the three-state protocol described in Section 7.1.2. However, given that epochs are not perfectly synchronized across distributed nodes, accurately mapping a function invocation to its corresponding epoch requires correlating both its timestamp and the specific node on which it executed. Consistent with the model formulation in Equation 6.6, a collection resides in its equilibrium position when its placement minimizes the system's potential energy. Consequently, for a given experiment, we define the system as being in an equilibrium state only when all managed collections have reached their respective equilibrium positions.

For deterministic experiments where the equilibrium state is known *a priori*, we classify epochs based on the system's placement state:

- **Unbalanced Epoch**: One or more collections remain outside their equilibrium position for the entire duration of the epoch.
- **Transition Epoch**: The system transitions into the equilibrium state at some point during the epoch.
- **Balanced Epoch**: All collections remain in their equilibrium positions for the entire duration of the epoch.

These epoch classifications are mutually exclusive; each epoch is assigned to exactly one category based on the system's state over the epoch duration. A transition from an unbalanced to a balanced state can occur either at the boundary between two epochs or within a single epoch. An experiment will therefore not necessarily contain Transition Epochs. The 'Transition Epoch' classification is used exclusively for cases where the system reaches equilibrium at some point during the epoch's time window. If the transition aligns with an epoch boundary, an Unbalanced Epoch is simply followed by a Balanced Epoch.

Based on these definitions, we refine our experimental objective to focus on assessing *con-*

vergence time, defined as the number of epochs required for the system to reach equilibrium, and analyzing *transient behaviors* by comparing user-perceived latency and message overhead during *Unbalanced Epochs* with the metrics observed during *Balanced Epochs*. This methodology serves a dual purpose: it allows us to quantify the system's end-to-end temporal behavior and to validate that the implemented data placement policy effectively aligns with the theoretical predictions of the mathematical model within a realistic environment.

The remainder of this section is organized as follows. We begin by detailing the Experimental Setup, describing the reference use case in terms of data types, actors, and client behaviors. Subsequently, we present five distinct experiments, each addressing a specific deterministic scenario. To ensure a systematic evaluation, every experiment is structured into three parts: a description of the Experimental Scenario, the Expected System Evolution derived from the theoretical model, and a detailed discussion of the Observed Results and Analysis.

8.5.1. Experimental Setup

Among the Motivating Scenarios introduced previously in Section 2.4.1, we selected the Smart Logistics use case as the baseline for our experimental evaluation. This scenario, which requires the orchestration of order distribution across a national territory, was chosen primarily due to its critical reliance on dynamic state migration, making it an ideal candidate for validating the effectiveness of our data placement protocols.

Within this use case, we identified three distinct actors relevant to our evaluation, mapping them to specific client classes:

- **Order Publishers:** Entities responsible for submitting new orders, representing either end-users or partner companies. They typically access the infrastructure via the cloud node.
- **Distribution Centers:** Entities responsible for managing orders and the vehicle fleet for deliveries. These clients are generally connected to proximal edge nodes.
- **Administrative Centers:** Decision-making hubs whose placement depends on the specific organizational structure. A vertical hierarchy implies a single, centralized administrative center hosted on the cloud node, whereas a horizontal structure employs multiple distributed centers located at the edge.

To ensure result interpretability, we enforce a static assignment policy where each client maps to a unique *Ermes Node*, ensuring a one-to-one correspondence. Furthermore, client

invocation rounds operate asynchronously relative to the internal epochs of the data placement protocol. This decoupling reflects a realistic operational scenario where user demand is independent of system control cycles.

To accommodate the heterogeneous storage requirements inherent in the Smart Logistics scenario, we categorize the application state into four distinct data types. Each type is defined by specific consistency guarantees and access patterns; depending on its logical scope, a data type maps either to a single collection or to a set of distinct collections:

- **Vehicle Fleet**: We assign a distinct, sequentially consistent collection to the vehicle fleet of each distribution center. Write operations, representing fleet status updates, originate locally from the corresponding distribution center, whereas read operations may be initiated from any node within the network.
- **Orders**: We aggregate all order information into a single, globally accessible, sequentially consistent collection. Clients at the cloud tier perform write operations to inject new orders, while distribution centers consume this data via read operations from their respective edge nodes.
- **Active Shipments**: We provision a dedicated, sequentially consistent collection for every active shipment. Both the source and destination distribution centers are granted read and write access, enabling them to monitor and update the shipment status from their respective edge nodes.
- **Performance Statistics**: We utilize a single, eventually consistent collection to serve as a centralized repository for performance metrics. All distribution centers perform write operations to log performance data and read operations to retrieve aggregated statistics.

We organize the client workload into 10 distinct rounds, each lasting 15 seconds. During each round, every client issues 50 invocations for each of the collection types it targets, maintaining a uniform injection rate of one request every 300 ms. These patterns follow the specific configurations detailed in the respective experiment descriptions. The workload remains invariant across all rounds: clients repeatedly execute the same set of actions, accessing the same subset of collections with consistent privileges throughout the duration of the experiment.

Table 8.1 summarizes these data categories and their relationships with the system actors.

Data Type	Cardinality	Consistency	Writers	Readers
Vehicle Fleet	1 per Dist. Center	Sequential	Local Dist. Center	Any client
Orders	1 Global	Sequential	Cloud Clients	All Dist. Centers
Active Shipments	1 per Shipment	Sequential	Sender & Receiver	Sender & Receiver
Performance Stats	1 Global	Eventual	All Dist. Centers	Any client

Table 8.1: Data Types, Consistency, and Access Patterns in the Smart Logistics Scenario.

To ensure reproducibility, the specific settings for the *Ermes Nodes* are provided in the Appendix, Table B.1. These parameters were applied across all subsequent experiments described in this evaluation.

8.5.2. Centralized Write, Distributed Read Pattern

Experimental Scenario

In this initial scenario, we model a **Top-Down Data Dissemination** scenario characterized by a centralized write and distributed read pattern. Specifically, a central entity publishes *Orders Data* to a single, sequentially consistent collection hosted at the cloud tier, while multiple distribution centers, represented by clients at the edge, consume this data. We configure the six clients participating in the experiment as follows:

- **Client 1 (W)**: One *Order Publisher* positioned on the **cloud node**, this client possesses **write** access to the *Orders Data* collection, simulating centralized data publication.
- **Clients 2–6 (R)**: Five *Distribution Centers* distributed across the five **edge nodes** (with one client per node), these clients hold **read-only** access to the *Orders Data* collection, representing distributed data consumption.

To implement these distinct access patterns, we employ two separate WASM functions, ensuring that each client group executes logic appropriate to its specific role.

Expected System Evolution

Consistent with the defined scenario, the system initializes the *Orders Data* collection and its associated metadata on the cloud node. In this unbalanced state, write operations from the cloud client execute locally with low latency, while read operations from the edge clients incur high latency due to remote data access.

At equilibrium, we expect our data placement algorithm to have positioned the data to optimize for this access pattern. Based on the results obtained by executing the same scenario in the mathematical model, the expected final state is that the leader replica of

the collection remains on the cloud node, co-located with the writer, while five read-only follower replicas are created, one on each of the edge nodes. In this balanced configuration, all function invocations become local to their respective clients, which should result in a significant reduction in user-perceived latency for the read-intensive clients at the edge.

Observed Results and Analysis

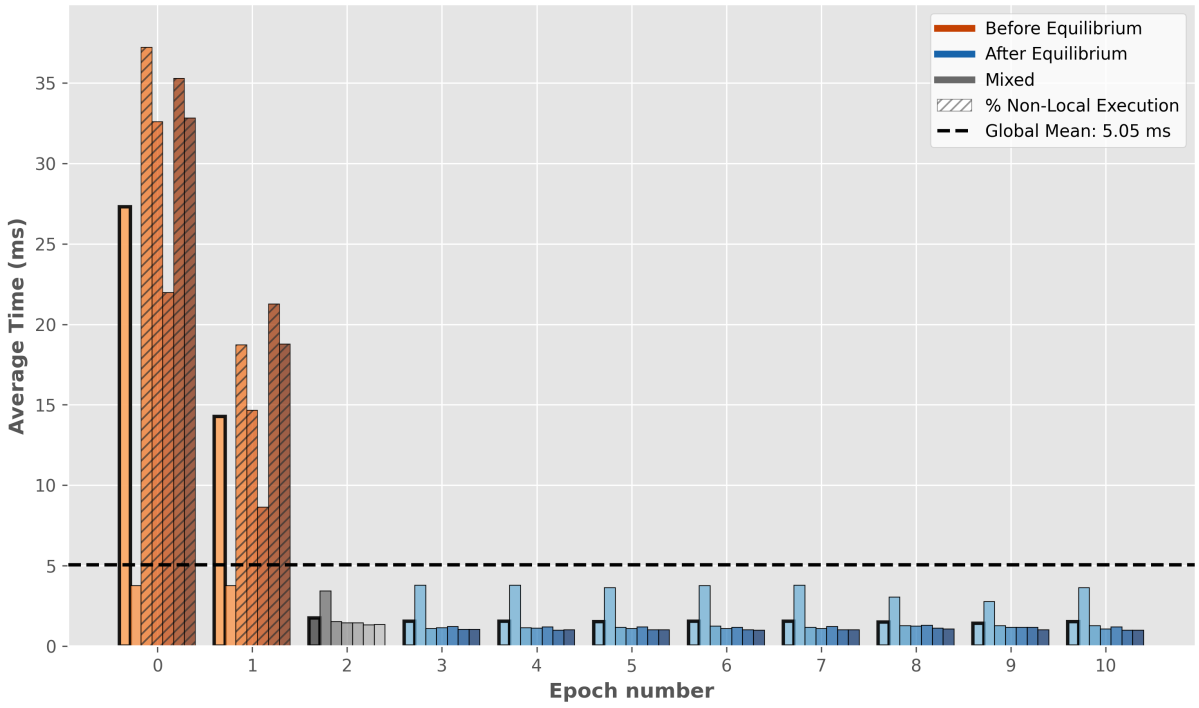


Figure 8.7: Top-down average latency.

Figure 8.7 presents the average function latency, aggregated by client and by epoch. The bolded bar for each epoch shows the system-wide average latency across all clients. In the first epoch (Unbalanced), write operations from the cloud client are local, while all read operations from the edge clients are remote, resulting in high average latency. Because our placement protocol allows a collection to move at most one tier per epoch, the system requires two epochs to reach equilibrium. In the second epoch, two follower replicas are created and placed on the Tier 2 nodes, reducing the average latency for read operations. The third epoch is a Transition Epoch, during which the follower replicas are migrated from the Tier 2 nodes to the Tier 1 edge nodes according to the implementation of the Physical model. The resulting balanced configuration coincides with the equilibrium state predicted by the theoretical model, confirming its empirical correctness. From this point

forward, the system is in a balanced state, and all read and write functions are executed locally, leading to a substantial reduction in latency.

Figure 8.8 illustrates the evolution of the workload distribution across the network hierarchy. Demonstrating that our placement strategy not only redistributes the computational load but also migrates routing and forwarding overhead. In the figure, the green sparklines represent the *Executed Functions*, corresponding to functions executed locally on a given node, while the blue sparklines denote the *Offloaded Functions*, comprising requests that are merely routed through the node to a different destination. Initially, the computational load is concentrated at the cloud node. As the system converges toward equilibrium, read-intensive operations are progressively offloaded to the edge nodes.

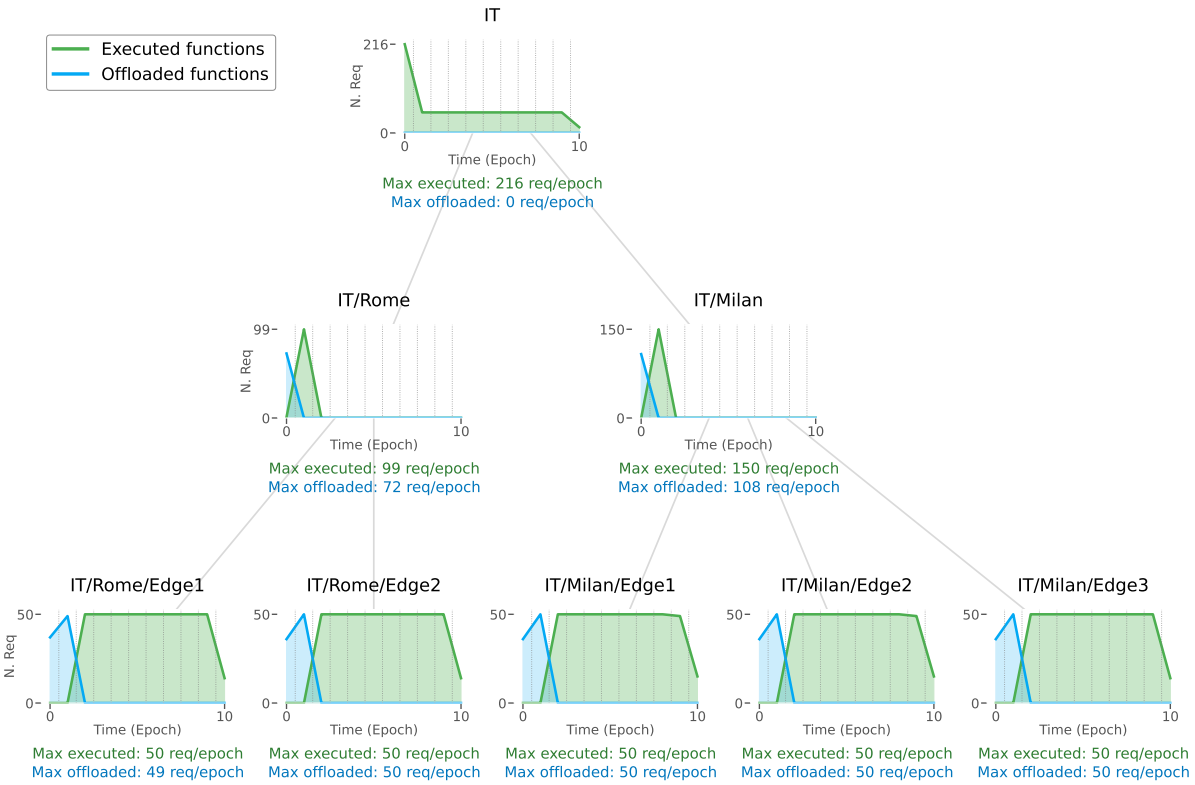


Figure 8.8: Top-down workload distribution.

Figure 8.9 provides a detailed breakdown of the latency distribution for each client throughout the experiment. The violin plot illustrates the latency range experienced by each client. Notably, the maximum latency observed for each client corresponds to the initial function invocation. This "cold start" incurs a one-time overhead that encompasses not only standard data lookup and execution, but also initial query resolution, the retrieval of function binaries and query templates, and the establishment of the necessary

HTTP connections.
The latency distribution reinforces this analysis, as the data points for each client are heavily concentrated near the minimum values, confirming that the high cold-start latency represents an outlier. Furthermore, this plot explicitly contrasts the average latencies of locally executed invocations against those served remotely, thereby quantifying the performance benefits yielded by our adaptive data placement strategy.

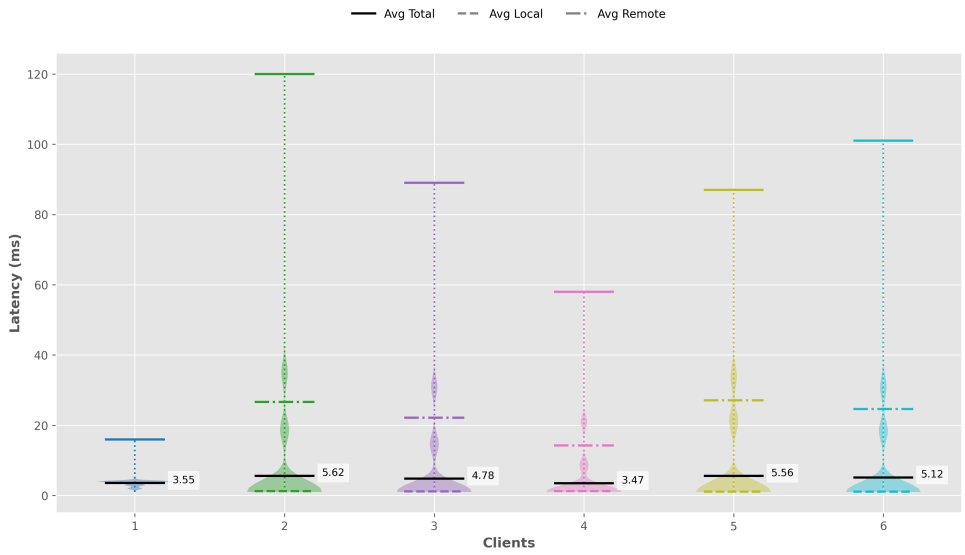
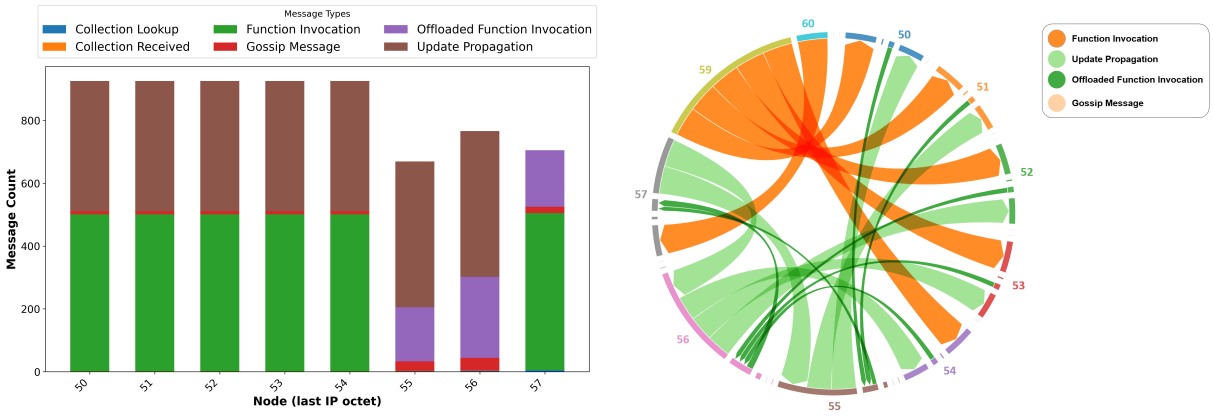


Figure 8.9: Violin graph of Top-down latency distribution by client.

In addition to latency, we analyze the network message overhead imposed by our framework. Figure 8.10(a) shows the total number of messages received by each node, categorized by type. As expected, the majority of messages received by the nodes directly connected to clients (Tier 1 and Tier 3) are function invocations, with each receiving 500 such messages. The primary source of overhead is update propagation messages, which flow from the cloud node to the edge nodes, as visualized in the chord diagram in Figure 8.10(b). A secondary source of overhead is the offloaded function invocations that are routed from the edge nodes to the cloud during the initial unbalanced epochs. In contrast, the gossip messages required by our distributed placement protocol constitute a minimal fraction of the total overhead. The low volume of these messages, at most 40 per node, and exactly 10 for each edge node over the entire experiment, demonstrates the information efficiency of our state-sharing mechanism.



(a) Received messages stacked bar chart (b) Messages chord diagram

Figure 8.10: Message overhead analysis for the Top-down scenario.

8.5.3. Centralized Read, Distributed Write Pattern

Experimental Scenario

We model a **Bottom-Up Data Aggregation** scenario where multiple distribution centers, represented by clients at the edge, publish updates to their respective *Vehicle Fleet Data Collections*. A central administrative client located at the cloud tier aggregates and reads data from all these collections to perform global analysis. The experiment involves six clients configured as follows:

- **Client 1 (R)**: One *Administrative Center* positioned on the **cloud node**, this client holds **read-only** access to all five *Vehicle Fleet Data Collections*, representing centralized data consumption.
- **Clients 2–6 (W)**: Five *Distribution Centers* distributed across the five **edge nodes** (one client per node), these clients possess **write** access to their dedicated *Vehicle Fleet Data Collection*, simulating distributed data publication.

By design, each writer targets a dedicated collection, in strict adherence to the system constraint limiting a single WASM function instance to at most one sequentially consistent collection. Consequently, this experimental scenario is inherently conflict-free: write contention is eliminated by the isolation of target collections, while potential read contention is resolved through the instantiation of new replicas. Furthermore, given that each read operation necessitates a distinct function invocation, the cloud client's aggregate invocation rate exceeds that of any individual edge client by a factor of five.

Expected System Evolution

Initially, each of the five *Vehicle Fleet Collections* is instantiated on its corresponding edge

node, co-located with its designated writer. In this unbalanced state, write operations initiated by edge clients execute locally with low latency. Conversely, read operations performed by the cloud client incur high latency, as each request must be routed to the specific edge node hosting the target collection.

At equilibrium, we anticipate that our placement algorithm will generate follower replicas to serve the read-intensive workload at the cloud. Consistent with the predictions derived from our mathematical model simulation, the expected final state retains the leader replica of each collection on its respective edge node, while migrating a read-only follower replica for each of the five collections to the cloud node. In this balanced configuration, all operations become local, yielding a significant reduction in latency for the cloud client.

Observed Results and Analysis

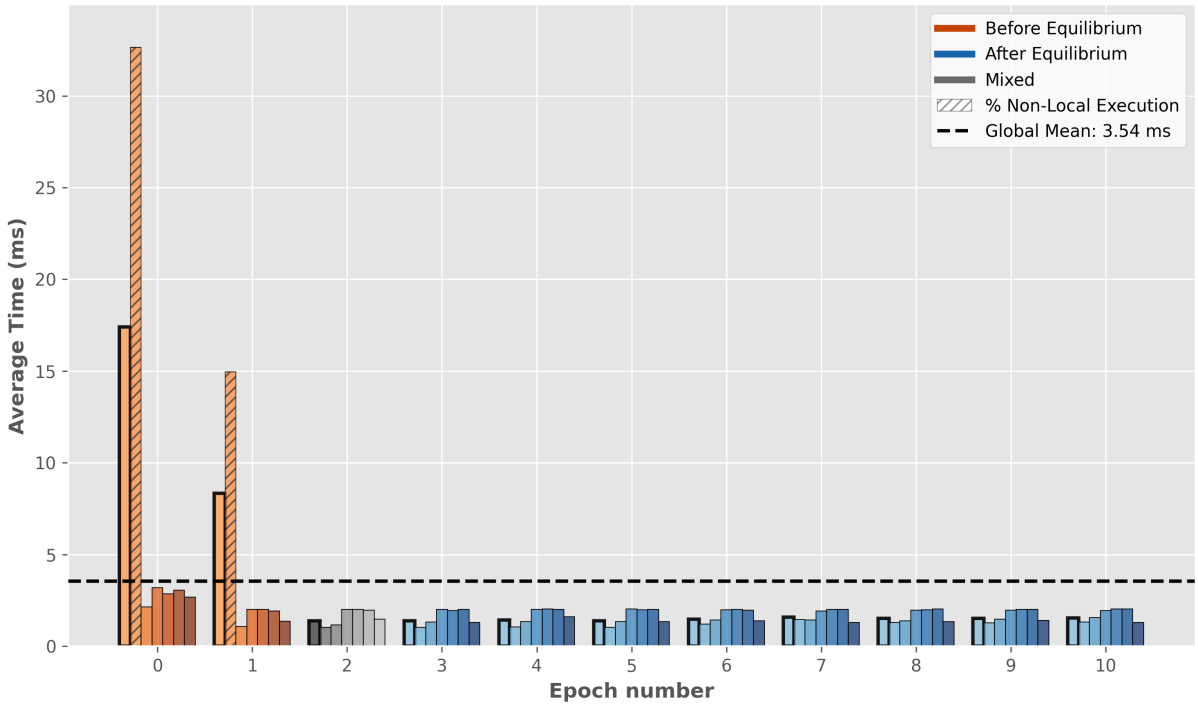


Figure 8.11: Bottom-up average latency.

Consistent with the first scenario, the system requires two epochs to reach equilibrium, as illustrated in Figure 8.11. However, a critical distinction in this configuration is that only operations originating from the cloud client are initially remote. The localization of edge write operations results in a lower system-wide average latency during the unbalanced epochs. Similarly, the workload distribution is initially confined to the edge nodes before

being partially offloaded to the cloud node. Figure 8.12 depicts the evolution of the workload across the network hierarchy; while the mechanism mirrors that of the previous scenario, the directionality of offloaded function invocations is inverted, flowing from the cloud toward the edge rather than from the edge to the cloud.

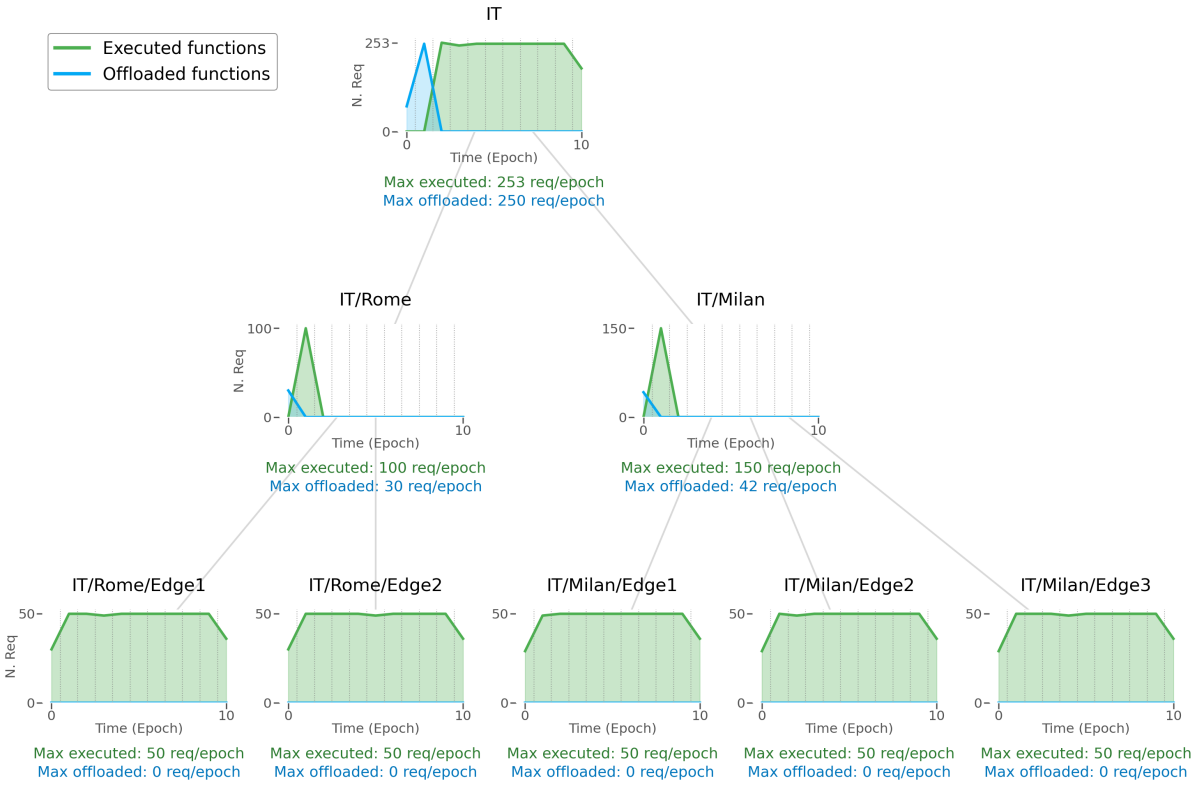
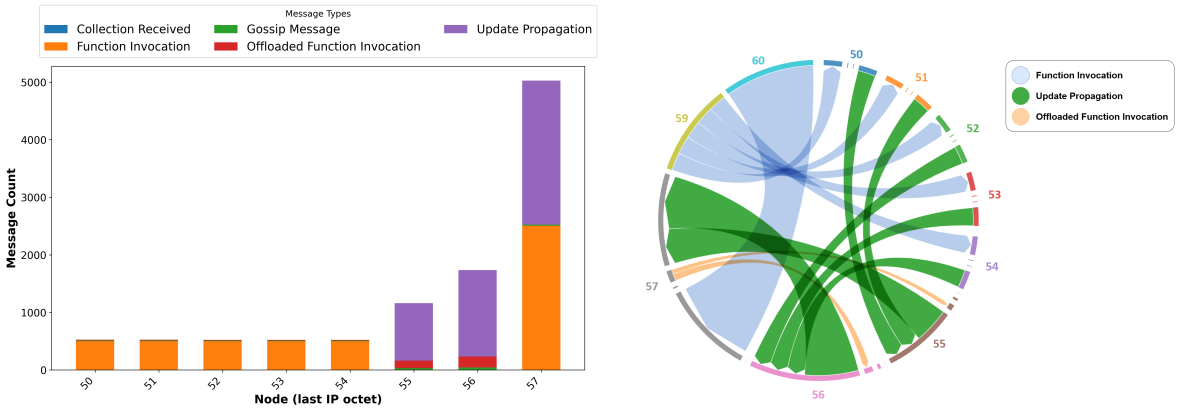


Figure 8.12: Bottom-up workload distribution.

A distinct characteristic of this scenario lies in the message overhead. Because this use case involves a higher frequency of write operations across multiple collections distributed among edge nodes, the volume of update propagation messages is significantly higher, as depicted in Figure 8.13(a). Furthermore, the directionality of these updates is inverted compared to the previous scenario; updates now originate at the edge nodes and propagate upward toward the cloud node.



(a) Received messages stacked bar chart

(b) Messages chord diagram

Figure 8.13: Message overhead analysis for the Bottom-up scenario.

8.5.4. Distributed Read and Write Pattern

Experimental Scenario

This scenario evaluates a distributed **Peer-to-Peer** data sharing pattern. We modify the previous configuration by replacing the single centralized administrative client with multiple administrative entities, each paired with a specific edge-based distribution center. In this setup, each distribution–administrator pair interacts exclusively through a dedicated and isolated *Vehicle Fleet Collection*. Consequently, there is no data contention or interference between pairs, rendering their operations fully independent. Given this isolation, we simplify the experimental evaluation to focus on a single representative pair of clients located on geographically distinct edge nodes. Within this pair, one client functions as the distribution center, performing write operations to the target *Vehicle Fleet Collection*, while the other acts as the administrative client, performing read operations. Following this simplification, the experiment involves two clients:

- **Client 1 (W)**: One *Distribution Center* positioned on a specific **edge node**, this client holds **write** access to its dedicated *Vehicle Fleet Data Collection*, simulating distributed data publication.
- **Client 2 (R)**: One *Administrative Center* Positioned on an **edge node** at the opposite end of the network topology, this client possesses **read-only** access to the same *Vehicle Fleet Data Collection*, representing distributed data consumption.

Expected System Evolution

Initially, the collection is instantiated on the edge node co-located with the distribution center client (Writer). In this unbalanced state, write operations execute locally with

minimal latency, whereas read operations initiated by the administrative client incur high latency as they must traverse the network hierarchy.

At equilibrium, consistent with the predictions of the mathematical model, we anticipate that our placement algorithm will establish a read-only follower replica of the collection on the administrative client's edge node. This configuration ensures that both read and write operations become local to their respective clients.

Observed Results and Analysis

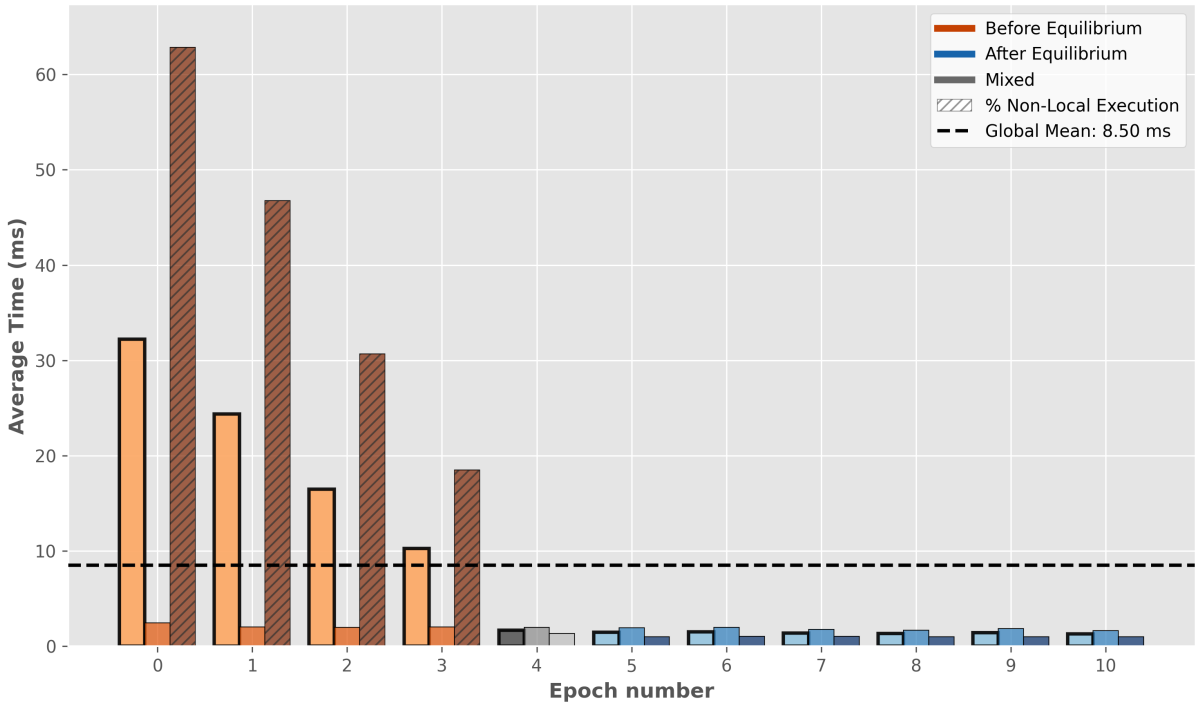


Figure 8.14: Peer-to-peer average latency.

Figure 8.14 presents the average function latency for the two clients. In the first epoch (Unbalanced), the writer's operations are local, while the reader's operations are remote. This scenario maximizes the initial network distance between the reader and the data, as the clients are placed on opposite sides of the network. Consequently, the system requires four epochs to converge to equilibrium. During this period, a follower replica is progressively migrated across the network, one tier per epoch, which gradually reduces the reader's perceived latency.

The extended convergence time results in a higher overall average latency compared to the previous two scenarios. This outcome suggests that in low-workload contexts, the duration

of the unbalanced phase is the primary determinant of average user-perceived latency. Furthermore, the experiment demonstrates that, provided the migration thresholds are met, the time to convergence is a direct function of the number of hops between the data's initial location and the remote reader.

Figure 8.15 illustrates the corresponding workload migration. As the follower replica moves from the writer's node toward the reader's node, the locus of execution for the read operations shifts across the network, residing at each intermediate tier for one epoch before reaching its final destination at the reader's edge node.

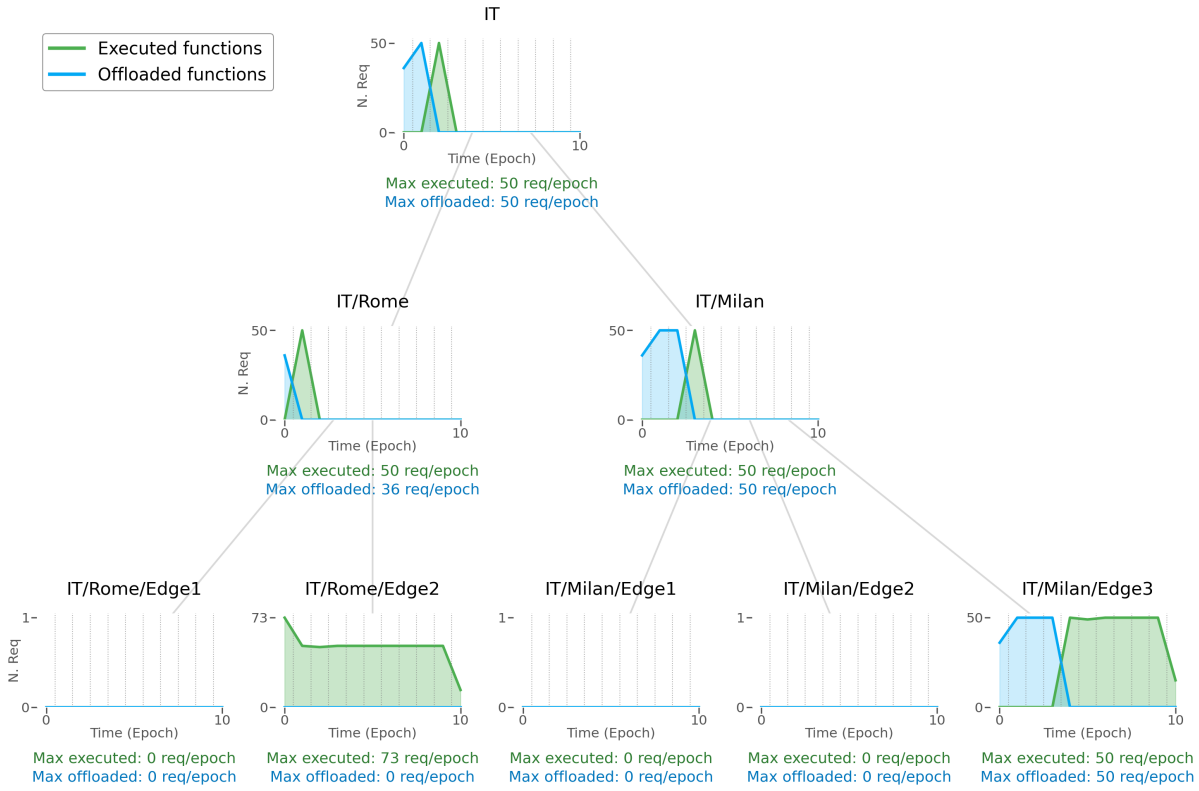


Figure 8.15: Peer-to-peer workload distribution.

Further analysis of the latency distribution, detailed in Figure 8.16, highlights the impact of the initial network distance. The figure indicates that, while minimum latency values remain comparable to those observed in previous experiments, the maximum latency for the remote reader is significantly higher. This increase stems from the "cold start" of the first invocation, which in this scenario encompasses not only remote query resolution (at the Cloud) and remote function execution across the entire network, but also the cumulative overhead of establishing HTTP handshakes as the request traverses multiple intermediate nodes. Similarly, the plot exhibits a wider distribution at higher latency

values for the remote reader compared to previous scenarios. Although the majority of invocations still cluster near the minimum latency, this wider distribution reflects the prolonged duration of the unbalanced phase as the replica migrates across the network.

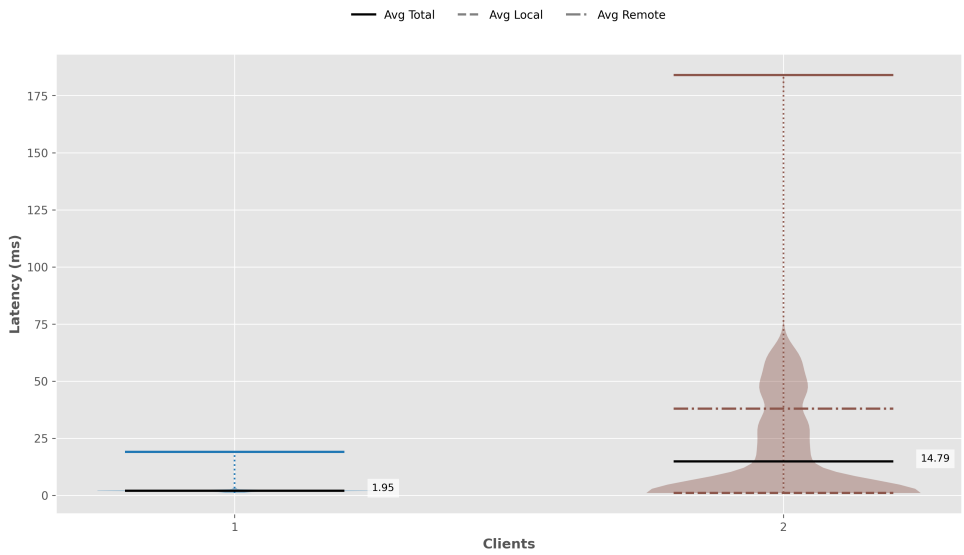


Figure 8.16: Violin graph of the Peer-to-peer latency distribution by client.

8.5.5. Competing Write Access and Leader Placement

Experimental Scenario

While the preceding scenarios involved conflict-free replica placement, where distinct client privileges (e.g., single writer, multiple readers) resulted in a deterministic optimal configuration, this experiment evaluates the framework's behavior under contention. Specifically, we analyze the **Conflict** scenario, wherein multiple actors simultaneously compete for the placement of the leader replica.

We model this by simulating an active shipment transfer between two distribution centers located on opposing edge nodes of the network. Both centers require read and write access to a single, shared collection responsible for tracking the shipment's status. The clients are configured as follows:

- **Client 1 (W & R)**: One *Distribution Center* positioned on a specific **edge node**. This client acts as **Shipment Initiator**, creating the *Active Shipment Collection* and maintaining **read and write** access.
- **Client 2 (W & R)**: One *Distribution Center* positioned on an **edge node** at the opposite end of the network topology. This client acts as **Shipment Receiver** and possesses **read and write** access to the same *Active Shipment Collection*.

Expected System Evolution

Initially, the collection is instantiated on the edge node associated with the shipment initiator. In this unbalanced state, the initiator's read and write operations execute locally with low latency, whereas the receiving center's operations are remote, incurring high network latency.

At equilibrium, consistent with the predictions of the mathematical model, we anticipate two distinct outcomes. First, to optimize local read access, a read-only follower replica should be instantiated on the edge node of each distribution center. Second, given that both clients exert a "write force" on the collection, we expect the leader replica to be positioned at a location that minimizes the aggregate write latency for both parties. In our testbed topology, where the clients are situated at opposite extremes of the network, this optimal location corresponds to the central cloud node. Consequently, in this equilibrium state, all read operations become local for both clients, while all write operations are executed remotely.

Observed Results and Analysis

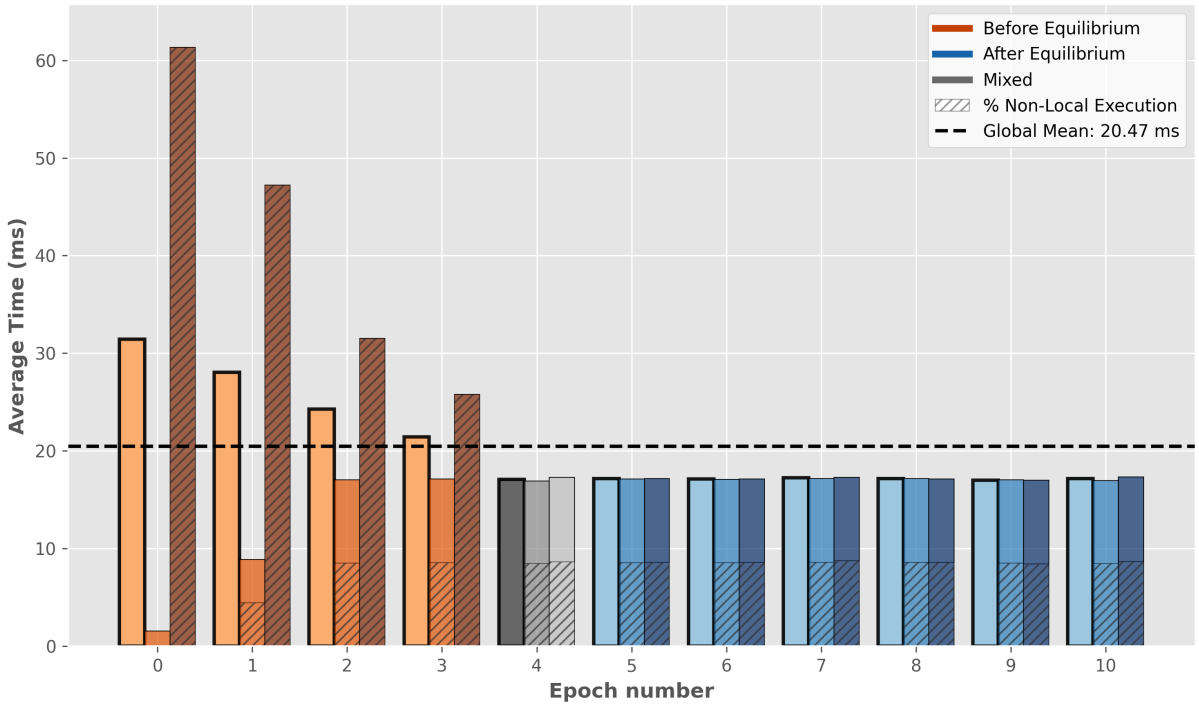


Figure 8.17: Conflict average latency.

Figure 8.17 presents the average function latency for the two competing clients. The

system's convergence to equilibrium unfolds in two distinct phases.

The first phase involves the migration of the leader replica. In the initial epoch, all invocations from the shipment initiator are local. Over the next two epochs, the leader replica migrates from the edge node up to the cloud node. As the leader migrates away from the initiator's node, a read-only follower replica is maintained in its place. Consequently, the initiator's read operations remain local and low-latency throughout this phase. The progressive increase in the initiator's average latency is therefore attributable exclusively to its write operations, which must now be directed to the increasingly distant leader. Conversely, the receiver's overall latency improves as the leader moves closer. At the end of the second epoch, the leader has reached its stable, central position in the cloud.

The second phase involves the creation of follower replicas. To address the remote reads for the receiver, our system creates a new follower replica from the leader in the cloud and migrates it down toward the receiver's edge node. This process takes an additional two epochs. The fourth epoch is a Transition Epoch, during which the follower replica arrives at the receiver's edge node. From this point forward, the system is in a balanced state. All read operations for both clients are now local, while all write operations are remote, directed to the leader in the cloud. This equilibrium state achieves a stable, lower average latency for both users by optimizing for local reads at the cost of remote writes.

Figure 8.18 illustrates the corresponding workload migration. Initially, computation is concentrated on the initiator's Edge node (IT/Rome/Edge2), while the Edge node co-located with the shipment receiver (IT/Milan/Edge3) must offload all functions. As the leader replica migrates up the hierarchy, the workload associated with write operations is progressively offloaded from the edge to the cloud. At equilibrium, the workload is distributed: 50% of the total invocations (all write operations) are executed on the cloud node, while the remaining 50% (all read operations) are split evenly between the two edge nodes. In this final state, the intermediate Tier 2 nodes no longer perform any direct computation and only handle the Offloaded Functions of routing the write requests to the cloud.

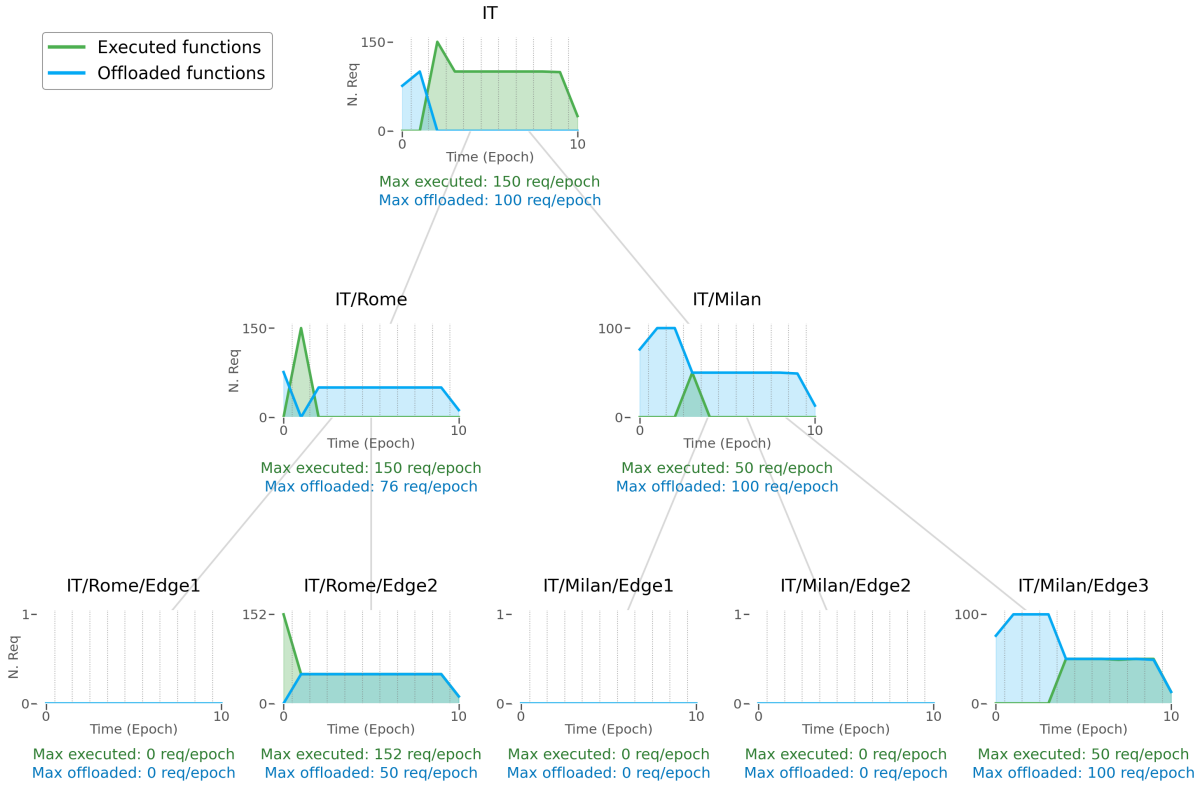


Figure 8.18: Conflict workload distribution.

Figure 8.19 provides a granular breakdown of the latency distribution for each user. Unlike the previous scenarios, the violin plots reveal two distinct latency clusters for each user. The first cluster, concentrated near the minimum latency, corresponds to local read operations, while the second, centered at a higher latency value, corresponds to remote write operations. This bimodal distribution, a direct result of the final equilibrium placement, clearly distinguishes between the two access types.

For the shipment initiator, the violin plot exhibits a wider base, reflecting the initial period of low-latency local writes. A key observation is the position of its write cluster: it is narrow and located near the maximum observed latency for this user. This demonstrates the performance degradation the initiator experiences for writes after the leader migrates to the cloud. In contrast, the write cluster for the shipment receiver is positioned significantly farther from its maximum latency and closer to its minimum, indicating a net performance improvement for its write operations at equilibrium. This highlights the trade-off inherent in the system: the initiator's write latency is intentionally increased to achieve a globally optimal placement that improves the average latency for both users.

The analysis of the maximum and minimum values in Figure 8.19 reveals a critical dif-

ference between the two clients. For the shipment initiator, the maximum latency corresponds to the remote write to the cloud at equilibrium. For the shipment receiver, however, the maximum latency remains the initial "cold start" invocation. This suggests that the one-time cost of the initial remote execution, which includes cross-network request routing and the cumulative overhead of multiple HTTP handshakes, exceeds the steady-state latency of writing to the cloud-based leader. This finding is significant because the initiator also incurs function and query fetching costs, albeit locally, indicating that network traversal is the dominant factor in the cold-start overhead for remote clients.

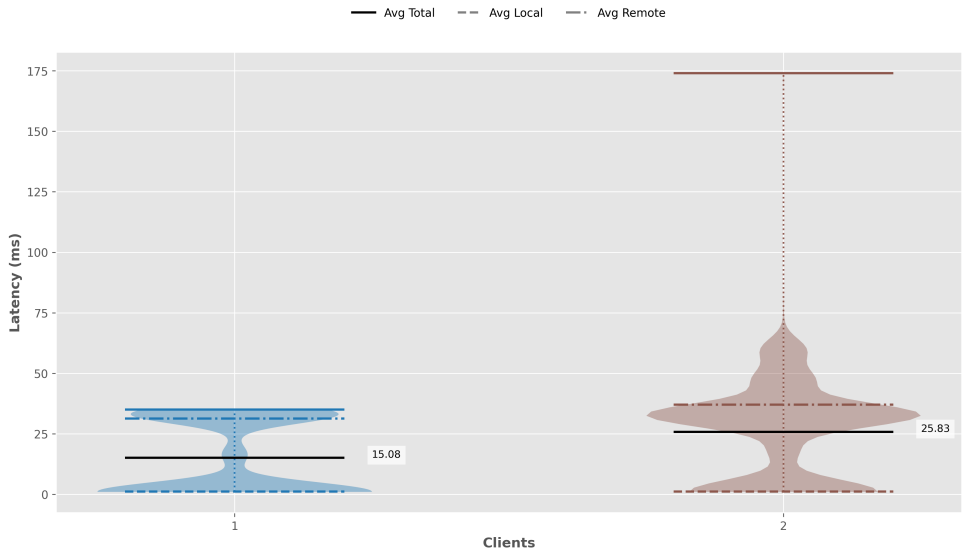
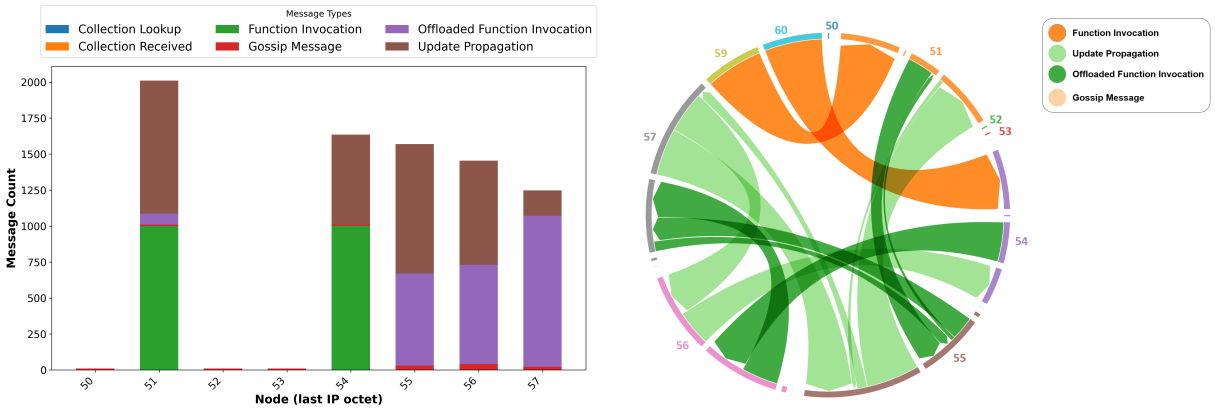


Figure 8.19: Violin graph of the Conflict latency distribution by user.

We also analyze the network message overhead for this scenario, as detailed in Figure 8.20. The stacked bar chart in Figure 8.20(a) confirms that the majority of the overhead consists of update propagation and offloaded function invocations. The distribution of these offloaded invocations changes as the system converges. Initially, the initiator's node (51) receives a small number of remote invocations from the receiver. At equilibrium, however, the cloud node (57) handles the majority of the offloaded workload, as it executes all remote write operations from both clients.

The flow of update propagation messages, visualized in the chord diagram in Figure 8.20(b), also reflects this convergence. During the unbalanced epochs, updates originate from the leader's transient location in the hierarchy and are propagated to the follower replicas. At equilibrium, the update flow stabilizes, originating exclusively from the leader on the cloud node (57) and propagating down to the two follower replicas on the edge nodes.



(a) Received messages stacked bar chart

(b) Messages chord diagram

Figure 8.20: Message overhead analysis for the conflict scenario.

8.5.6. Evaluating Mixed-Consistency Access Patterns

Experimental Scenario

This experiment expands upon the previous configuration to evaluate system performance when functions concurrently access collections with heterogeneous consistency guarantees. We modify the setup by introducing an eventually consistent *Performance Statistics Collection* alongside the original sequentially consistent *Shipment Collection*, establishing a **Two Collection Conflict** scenario. We define two distinct functions: the first performs write operations on both collections, while the second reads from the *Shipment Collection* and writes to the *Performance Statistics Collection*. The client configuration is as follows:

- **Client 1 (W & R)**: One *Distribution Center* positioned on a specific **edge node**. This client acts as **Shipment Initiator**, creating both the *Active Shipment Collection* and the *Performance Statistics Collection* and maintaining **read and write** access to both.
- **Client 2 (W & R)**: One *Distribution Center* positioned on an **edge node** at the opposite end of the network topology. This client acts as **Shipment Receiver** and possesses **read and write** access to both collections.

Expected System Evolution

Consistent with the preceding scenario, we expect the system to initialize both collections on the edge node associated with the shipment initiator. In this unbalanced state, the initiator's read and write operations execute locally with minimal latency, whereas the receiver's operations are remote, incurring significant network latency.

The central hypothesis of this experiment is that the placement of a function is dictated

by its strongest consistency requirement. Consequently, the placement of the sequentially consistent collection's leader will determine the execution location for any function that writes to it. At equilibrium, we expect the same placement for the *Shipment Collection* as in the previous scenario: the leader migrates to the central cloud node, while follower replicas are placed on each edge node.

This primary placement decision influences the replication of the eventually consistent data. Because the function that writes to the sequential collection must execute on the cloud, its write to the *Performance Statistics Collection* will also originate from the cloud, creating demand for a replica there. The other function, which only reads the sequential collection, will execute on the edge nodes, requiring local replicas of the eventually consistent data as well. We therefore expect the *Performance Statistics Collection* to be replicated on both edge nodes and the cloud node. At equilibrium, functions that only read the sequential collection will execute entirely locally at the edge. In contrast, any function performing a write to the sequential collection will be executed remotely on the cloud, making that invocation remote for the edge client, even though the subsequent data accesses from the cloud-based function will be local.

Observed Results and Analysis

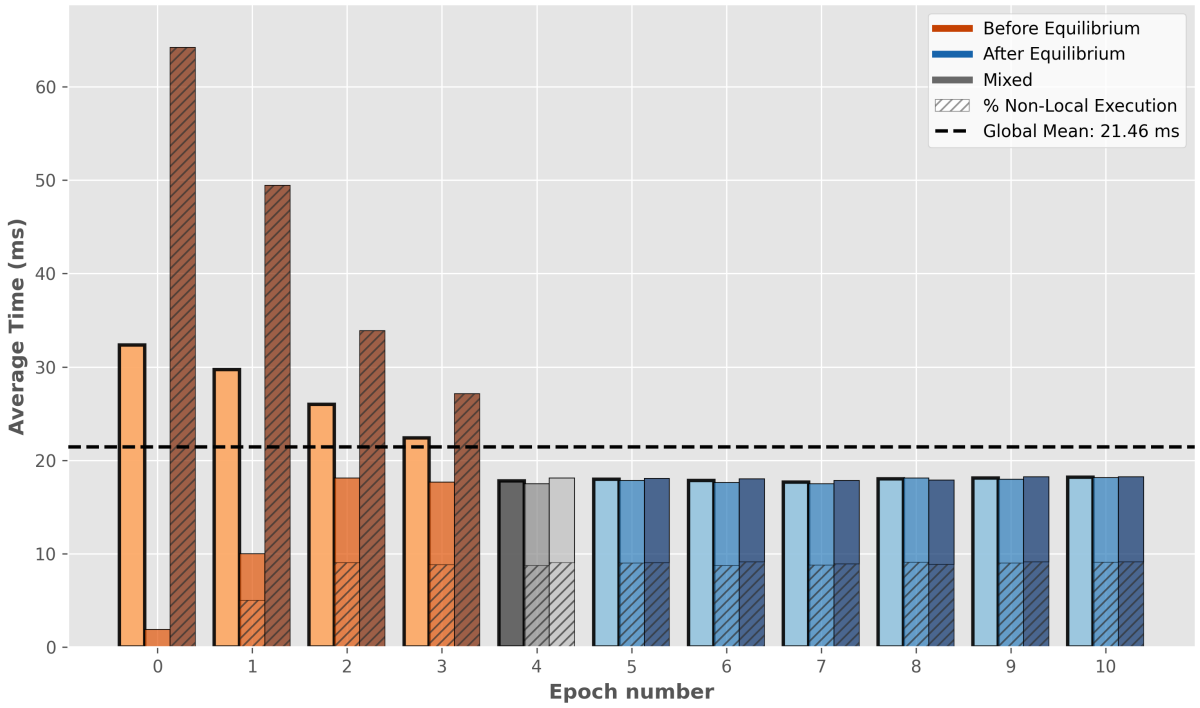


Figure 8.21: Two-collection-conflict average latency.

The results of this experiment, shown in Figure 8.21 and Figure 8.22, confirm our central hypothesis. Both the evolution of latency over time and the workload migration pattern are nearly identical to those observed in the previous Conflict scenario. The leader of the sequentially consistent collection migrates from the edge to the cloud, leaving a follower replica behind, and a second follower replica is subsequently created and migrated to the other edge node. The fact that all functions now also write to an eventually consistent collection does not alter this placement behavior, demonstrating that the strongest consistency requirement dictates the function's execution location.

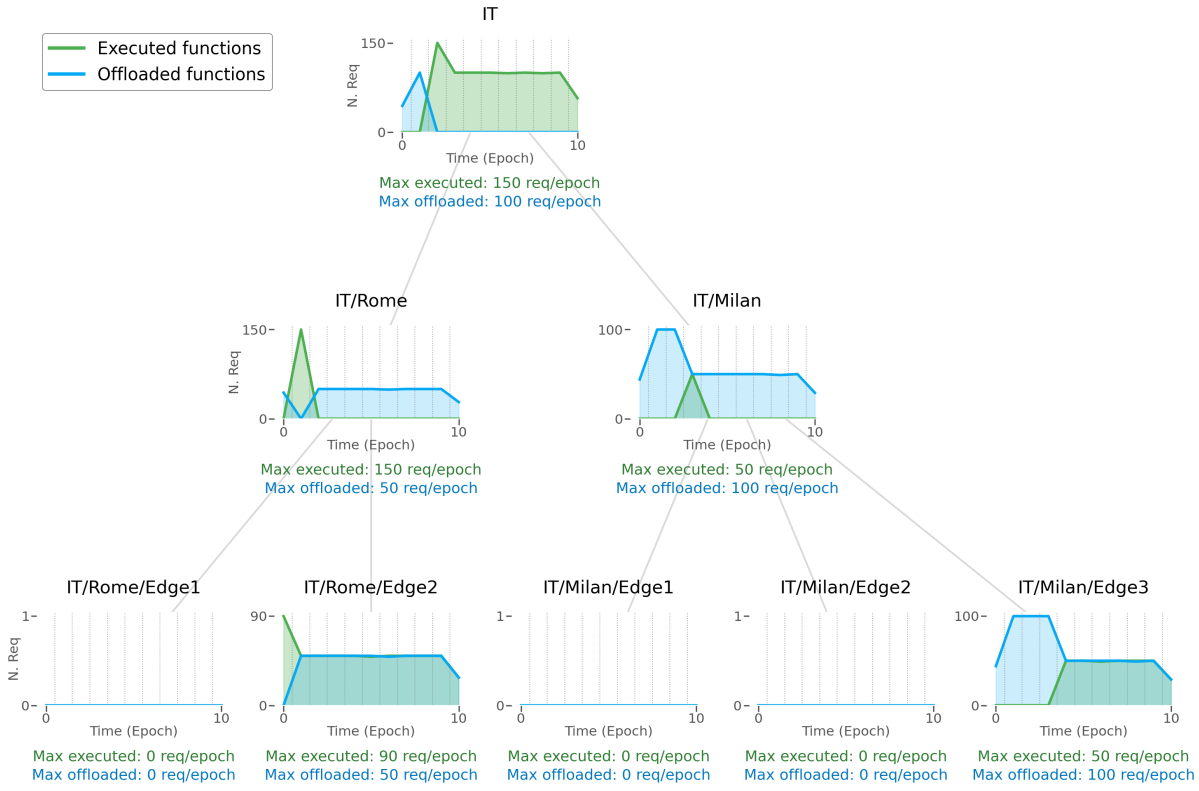
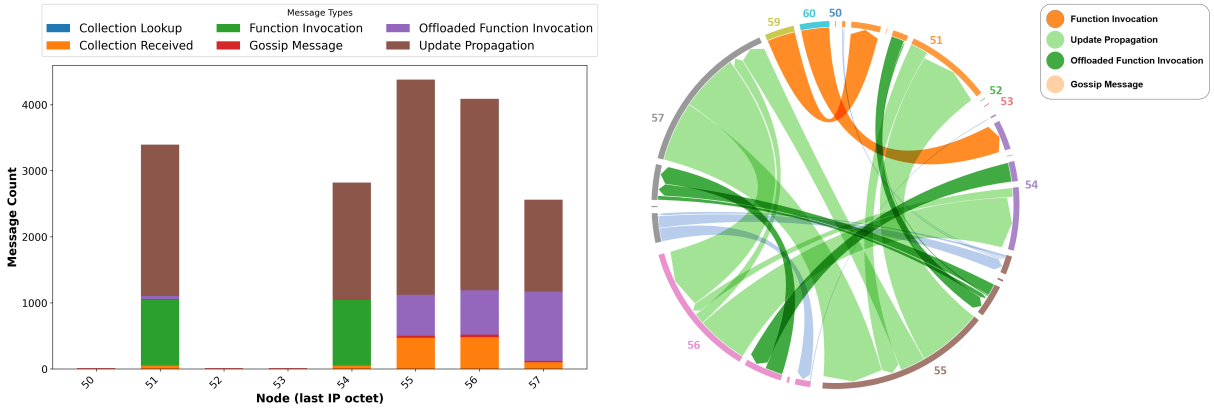


Figure 8.22: Two-collection-conflict workload distribution.

The primary difference from the previous scenario lies in the network message overhead. While the function invocation pattern remains unchanged, resulting in a similar number of offloaded function messages, the volume and direction of update propagation traffic are significantly different, as shown in Figure 8.23. At equilibrium, there is now a bottom-up flow of updates in addition to the top-down flow. This is because the function that reads the sequential collection executes locally on the edge nodes, and its writes to the eventually consistent collection generate updates that propagate up the hierarchy.

This outcome also highlights the different placement strategies for the two consistency

levels. The physical model for the sequential collection is demand-driven, placing replicas only where required (the cloud and two edge nodes). In contrast, the "always-copy" policy for the eventually consistent collection is more opportunistic: because update messages for the sequential collection pass through the intermediate Tier 2 nodes, our policy also creates replicas on them. Consequently, at equilibrium, the Tier 2 nodes (55 and 56) not only route update messages but also host a replica of the eventually consistent collection and must apply updates locally, a behavior not observed in the previous scenario.



(a) Received messages stacked bar chart (b) Messages chord diagram

Figure 8.23: Message overhead analysis for the two-collection-conflict scenario.

8.6. Ablation Study under Stochastic Access Patterns

The primary objective of these experiments is to assess the system's performance, scalability, and robustness under stochastic operational conditions and varying workload intensities. Unlike the deterministic use cases analyzed in the previous section, these experiments introduce randomized data access patterns and function invocations to simulate realistic scenarios, thereby testing the architecture's adaptability in dynamic environments. We adopt a systematic ablation methodology: initially, we establish a performance baseline using the complete Ermes framework. Subsequently, we selectively disable specific architectural mechanisms to emulate the functional constraints typical of alternative state-of-the-art solutions. This comparative strategy enables a direct performance assessment against competitor-like configurations and allows us to isolate and quantify the contribution of individual design choices to critical system metrics.

The remainder of this section is organized as follows. We begin by detailing the experimental setup, describing the generation of stochastic scenarios for user and collection placement as well as user-collection associations. Subsequently, we present three experiments. We start by assessing the performance of Ermes under stochastic workloads,

varying the number of users and collections. Then, we disable fundamental features and compare the results against the Ermes baseline to evaluate the contribution of individual components to the overall performance.

8.6.1. Experimental Setup

We generate experimental scenarios by randomly sampling configurations within the user-collection placement space, progressively increasing the number of users and collections to evaluate system scalability. We place users exclusively on edge nodes, selecting locations via a uniform distribution. Conversely, collections may initially reside on any node within the Ermes network topology; their specific placement is determined by sampling from a Gaussian distribution centered on the network root. We intentionally assigns a significantly higher probability to upper-tier nodes, thereby concentrating initial data placement within the cloud and core infrastructure rather than at the network edge.

To define the workload, we assign a set of target collections and an invocation pattern to each user. Specifically, each user is granted access to a set of up to 25 target collections, selected via a uniform distribution. The user's function invocations then target these collections, also following a uniform distribution, which permits repeated access to the same collection. The access pattern for all users follows a probabilistic model with an 80% read and 20% write operation ratio. We chose this distribution to reflect the asymmetric nature of typical edge-cloud workloads, where data consumption operations significantly outnumber state mutations.

Each client performs a sequence of 25 function invocations targeting the assigned collections. We distribute these operations over a 15-second interval, applying a randomized start delay to each user to mitigate synchronization artifacts. To ensure the system has sufficient time to converge to the expected equilibrium, we repeat this execution pattern identically 10 times, consistent with the methodology established in our previous experiments. Furthermore, we execute each experimental configuration multiple times using distinct random seeds to minimize statistical variance resulting from the stochastic nature of the scenario generation.

As the expected number of collections accessed by a single user increases (up to a maximum of 25), the effective force exerted by that user on any individual collection decreases. Consequently, we must lower the physical model thresholds in proportion to the increasing number of collections to ensure appropriate migration behavior. We express this dynamic adjustment using the following formulas:

Defining C the total number of collections in the system

Defining U the total number of users in the system

Defining max_u the maximum number of user involved in te system

$$\text{Leader\_threshold} = \frac{450 \cdot \min(C, 4)}{\min(C, 25)} \cdot (mU + q)$$
$$\text{Follower\_threshold} = \frac{150 \cdot \min(C, 4)}{\min(C, 25)} \cdot (mU + q)$$

Where:

$$m = \frac{1}{max\_u + 200}$$
$$q = \frac{200}{max\_u + 200}$$

Table B.2, in the Appendix, details the specific configuration parameters for the *Ermes Nodes* used in these experiments, highlighting deviations from previous experimental setups.

8.6.2. Ermes Framework Scalability Baseline

Experimental Scenario

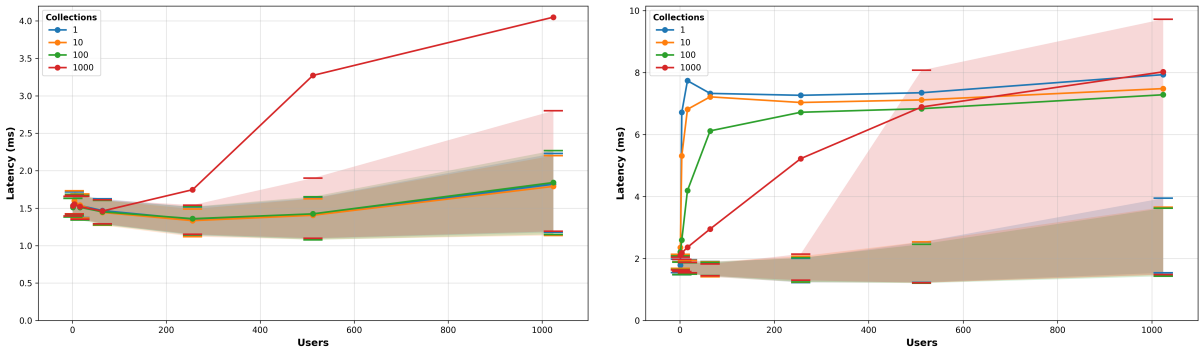
To establish a performance baseline, we evaluate the Ermes framework in its default, fully-featured configuration. We execute these experiments using the stochastic scenarios defined in the experimental setup, testing under both of our supported consistency models. This comparative analysis allows us to empirically quantify the performance trade-offs between the two models, focusing on user-perceived average latency and the system's transient behavior during convergence.

Observed Results and Analysis

Figure 8.24 illustrates the scalability of system performance at steady state (i.e., after all collections have reached their equilibrium positions), measured in terms of user-perceived average latency and interquartile range (IQR). The results are presented as a function of the number of active users and the total number of collections.

For eventually consistent collections, as shown in Figure 8.24(a), we observe consistently low latencies, typically in the 1-2 ms range, for scenarios with up to 100 collections, even with as many as 1024 concurrent users. This low latency is a direct consequence

of our "always-copy" policy, which aggressively replicates data and allows writes to be served locally by any replica. However, a significant deviation from this trend occurs in the scenario with 1000 collections. When the user count exceeds 256 in this configuration, both the average latency and the upper bound of the IQR begin to increase. This performance degradation is caused by memory pressure on the edge nodes; the aggressive replication policy exhausts the available RAM on the Tier 1 nodes, preventing the placement of all required collections at the user's entry point. As a result, some collections remain at Tier 2, forcing function execution to higher levels of the hierarchy and thereby incurring additional network latency.



(a) Eventual Consistency (b) Sequential Consistency

Figure 8.24: Scalability of user-perceived average latency in Ermes with stochastic workloads (default configuration).

As anticipated, operations on sequentially consistent collections (Figure 8.24(b)) exhibit higher average latency than their eventually consistent counterparts. This overhead stems directly from the stricter consistency model, which necessitates the serialization of all write operations through a single leader replica. When geographically distributed users contend for the same collection, this centralization introduces significant network latency, as the leader is typically placed at higher network tiers. Consequently, as the number of users increases, the average latency grows, and this effect is inversely proportional to the number of collections; with fewer collections, the probability of write conflicts on any single collection is higher. In these experiments, the latency impact of leader centralization is the dominant performance factor. The effects of memory saturation only become visible as a widening of the IQR when the user count exceeds 512, but they do not significantly alter the average latency trend.

The performance degradation observed in the 1000-collection experiments is a direct consequence of the aggressive replication policy, which aims to distribute the maximum number of collections in close proximity to the users. Nevertheless, for both consistency

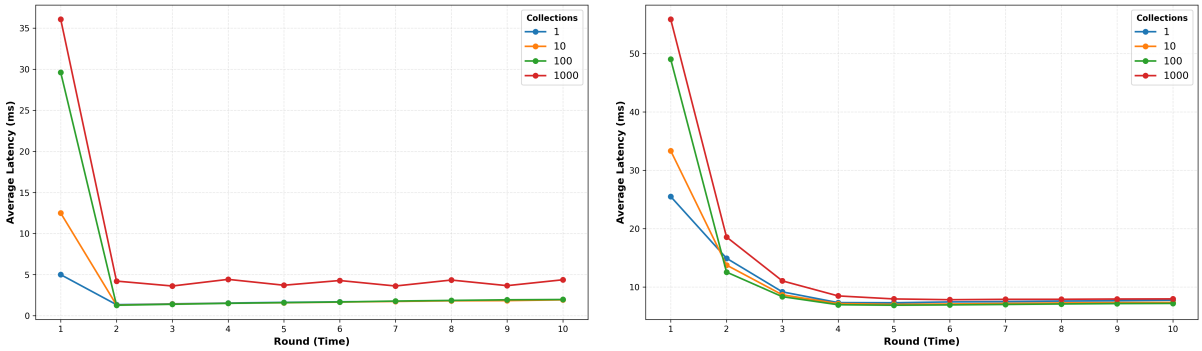
policies, the measured average latency remains limited, especially when compared to the ablated scenarios discussed in the following sections.

To investigate the impact of different consistency models on the system's transient behavior, we fixed the number of users at 1024 and varied the number of collections, sampling the average latency every 15 seconds. The results are shown in Figure 8.25.

For scenarios with up to 100 collections (Figure 8.25(a)), we observe that the eventually consistent collections reach their optimal placement within the first round; the second sampled measurement already reflects the steady-state latency. In contrast, the sequentially consistent collections (Figure 8.25(b)) exhibit a logarithmic convergence pattern. Their latency curves monotonically decrease until stabilizing at a steady-state value around the fourth round. As expected, the slope of this logarithmic curve is steeper for a larger number of collections, as the lower probability of write conflicts on any single collection allows the system to converge more quickly.

The results for the 1000-collection scenarios require a separate analysis. For eventually consistent collections, the average latency is not only higher but also fails to stabilize, instead oscillating just below 5ms. This behavior is a direct consequence of the interaction between our "always-copy" policy and the LRU-based memory management. At the end of each epoch, edge nodes under high memory pressure attempt to offload a portion of their collections to their parent to remain below the configured memory "safe level". However, the "always-copy" policy simultaneously attempts to create new replicas on those same edge nodes in response to user invocations. This conflict causes some collections to be repeatedly moved between Tier 1 and Tier 2, leading to the observed latency fluctuations at steady state.

Similarly, the curve for 1000 sequentially consistent collections not only converges to a higher latency value but also requires more rounds to do so. This slower convergence is a result of the increased memory pressure on the edge nodes. As modeled by the hydrostatic potential energy (Equation 6.5), higher memory pressure increases the "energy" required to pull a collection down to a Tier 1 node. Consequently, more user invocations, and thus more epochs, are needed to generate sufficient force to trigger the migration and reach equilibrium.



(a) Eventual Consistency (b) Sequential Consistency

Figure 8.25: Latency Time behaviour with 1024 users (default configuration).

The performance degradation and transient effects observed in the 1000-collection experiments stem directly from the placement policies and their interaction with memory pressure when LRU capacity approaches hardware limits. In this configuration, Tier 1 memory is nearly saturated, forcing the system to place certain collections on Tier 2 nodes, which retain available capacity. Even in the presence of these transient phenomena, the performance degradation is not significant and does not constitute a fundamental scalability limit of the proposed solution; measured average latencies remain below 10 ms for both consistency policies. Comparing these values with the ablated configurations (described in Sections 8.6.3 and 8.6.4), which exhibit latencies in the order of tens of milliseconds ($\approx$ 25–35 ms), confirms that our architecture maintains superior performance under heavier workloads. The apparent scalability constraints are primarily artifacts of the specific network structure rather than intrinsic architectural flaws.

8.6.3. Ablation of Local Persistence: Evaluating Stateless Architectures

Experimental Scenario

To benchmark our system against traditional architectures, we compare the performance of the Ermes baseline configuration with a stateless FaaS model characterized by externalized persistent storage. To minimize confounding variables and isolate the specific impact of data distribution (stateful versus stateless) on latency, we emulated a stateless architecture using the Ermes framework itself. We achieved this by configuring all *Ermes Nodes* within the hierarchy to bypass their local Redis instances, directing all persistence operations to a dedicated node positioned in close proximity to the cloud layer (2ms RTT) that acts as a centralized storage repository. The resulting network topology is illustrated in Figure 8.26. In this configuration, functions are invariably executed on the edge node

closest to the user, in contrast to the data-locality approach employed in the default configuration, and must access persistent data remotely via the centralized database.

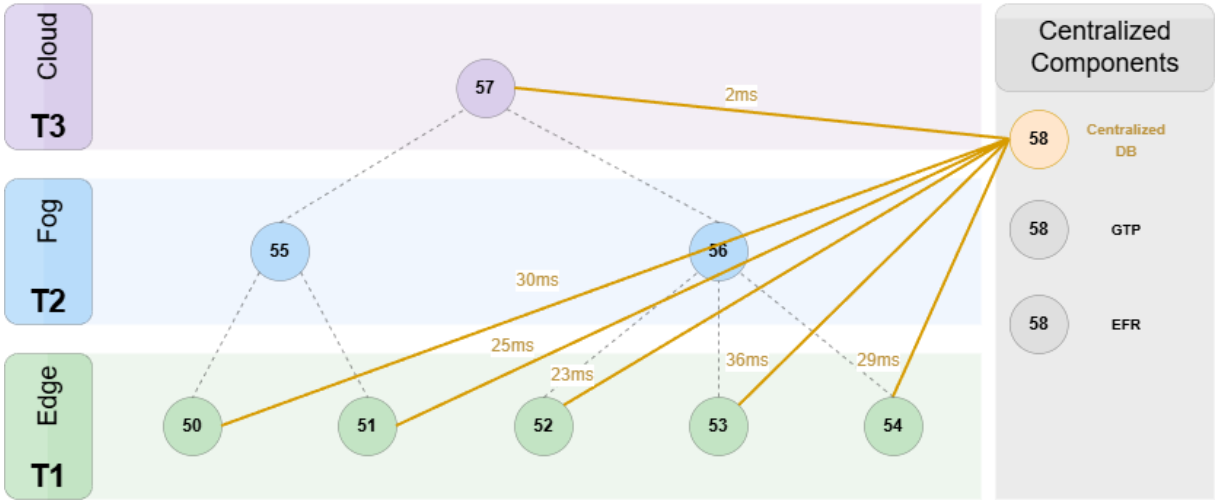
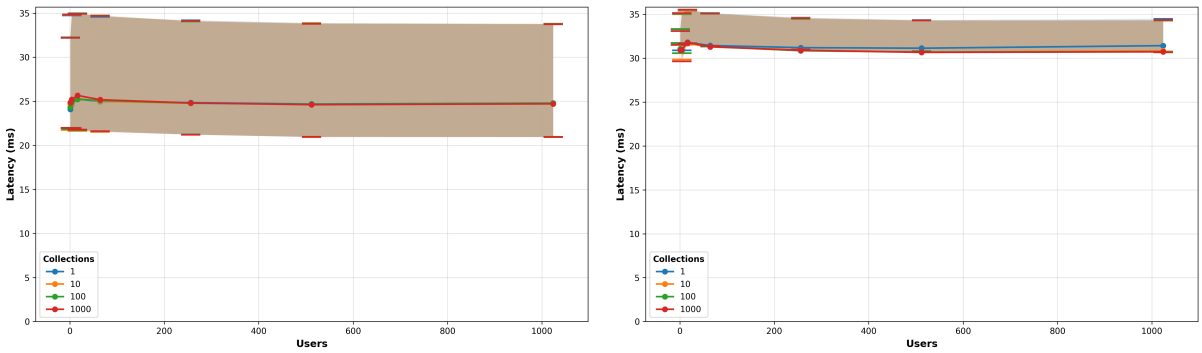


Figure 8.26: Ermes Stateless Network.

In this centralized configuration, Ermes maintains the consistency and session guarantees defined in the default setup. Although enforcing sequential consistency is inherently simpler in a centralized environment, as it eliminates the need for complex replica synchronization, we continue to utilize vector clocks to uphold client-centric session guarantees. This architecture imposes distinct latency penalties depending on the operation type. Crucially, all read operations are executed synchronously, as function execution strictly depends on the retrieval of remote data, regardless of the consistency model. The handling of write operations, however, varies: writes to eventually consistent collections are performed asynchronously, effectively masking network latency, whereas writes to sequentially consistent collections require synchronous acknowledgement to retrieve and return the updated vector clock to the client, thereby incurring the full network round-trip cost.

Observed Results and Analysis

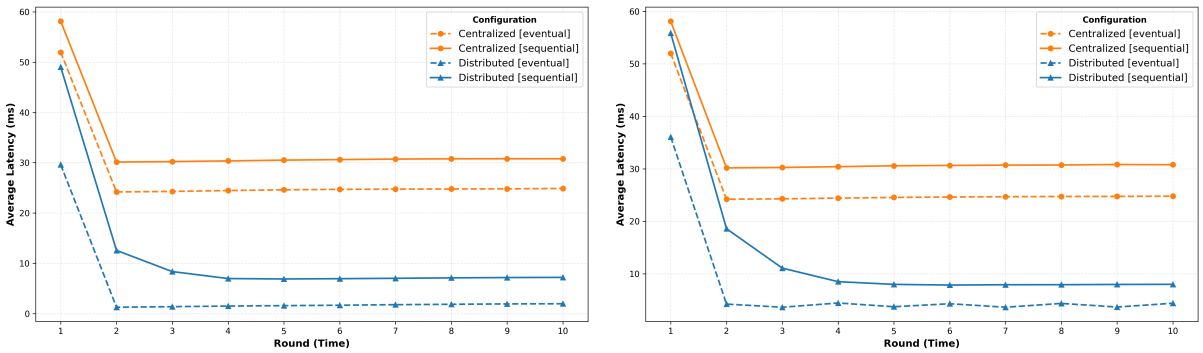


(a) Eventual Consistency (b) Sequential Consistency

Figure 8.27: Scalability of user-perceived average latency in an emulated stateless FaaS model, where persistent storage is externalized.

Figure 8.27 depicts the user-perceived average latency and the corresponding Interquartile Range (IQR) at steady state for the emulated stateless architecture. The results indicate a disparity in average latency between the two consistency models, with Eventual Consistency exhibiting lower values and a wider IQR compared to Sequentially Consistent collections. This characteristic is a direct consequence of the asynchronous handling of write operations, which constitute 20% of the workload for collections with weaker consistency guarantees. Furthermore, a comparison of these results with the Ermes baseline shown in Figure 8.24 demonstrates that our proposed distributed architecture achieves a substantial performance advantage over this centralized baseline. Indeed, for both consistency models, the system-wide average latency is significantly higher in the centralized scenario.

This performance gap is further substantiated by the IQR analysis. Across all tested combinations of user and collection counts, the lower bound (25th percentile) of the centralized configuration's IQR consistently exceeds the upper bound (75th percentile) of the distributed configuration's IQR. This indicates a fundamental statistical divergence between the two latency distributions; in effect, the fastest 25% of operations in the centralized system are slower than the slowest 25% of operations in our distributed system. This clear separation confirms that the performance gains offered by our data-local design are not limited to average-case improvements but represent a systematic, architecture-level enhancement across the entire latency profile.



(a) 100 collections (b) 1000 collections

Figure 8.28: Latency Time behavior with 1024 users.

An analysis of the system's transient behavior further confirms this performance advantage. Figure 8.28 compares the temporal evolution of latency for the Ermes baseline configuration and the emulated stateless system, using a fixed workload of 1024 users. Our proposed framework consistently outperforms the centralized baseline. Notably, even before reaching its optimal placement, the latency for sequentially consistent collections in our distributed system is lower than the steady-state latency for eventually consistent collections in the centralized model. This holds true for both the 100-collection scenario (Figure 8.28(a)) and the 1000-collection scenario (Figure 8.28(b)), where edge node memory saturation occurs.

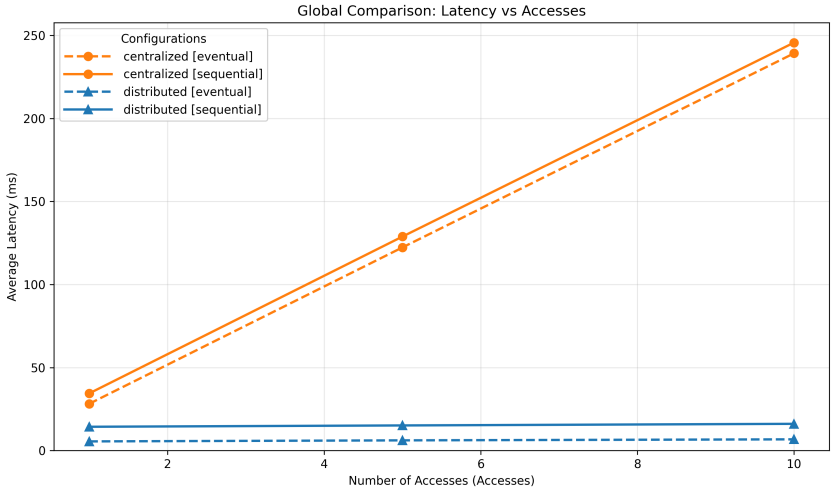


Figure 8.29: Global average latency comparison varying number of accesses.

The analysis thus far has focused on scenarios where each function performs a single data access. To evaluate the performance under more data-intensive workloads, we now analyze

the case where a single function execution involves multiple accesses to the same collection. For this experiment, we used a fixed configuration of 64 users and 100 collections. We confirmed through supplementary tests that this configuration is representative, as varying these parameters yielded qualitatively similar results, indicating that the observed behavior is robust to changes in workload intensity and data partitioning.

As illustrated in Figure 8.29, the performance of the two architectures diverges significantly under this multi-access pattern. The centralized architecture is penalized by the cumulative latency of repeated remote round-trips, as each access within the function incurs the full network overhead. In contrast, our distributed solution exploits data locality: once the collection is co-located with the function, all subsequent accesses within the same invocation are served locally. Consequently, as the number of accesses per function increases, the total latency for the centralized architecture grows linearly, while it remains nearly constant for our distributed system. This widening performance gap underscores the scalability benefits of our approach, particularly for stateful functions and data-intensive workloads characterized by iterative processing or fine-grained data manipulation.

8.6.4. Ablation of User-Side Caches

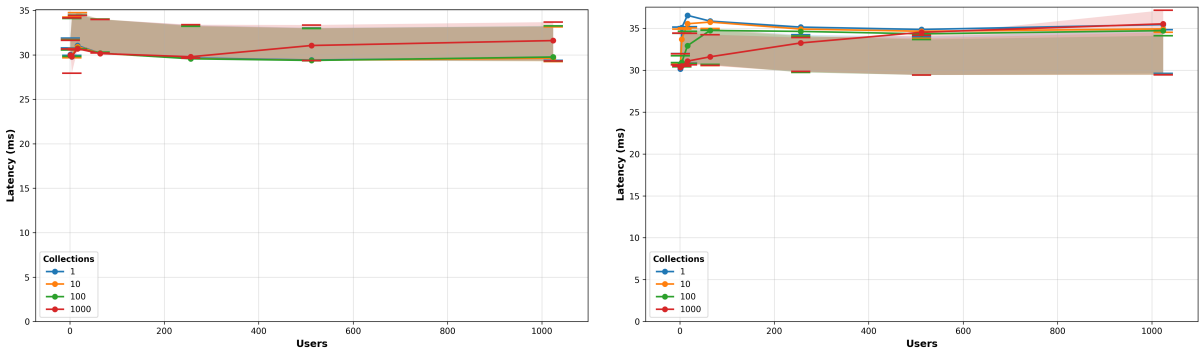
Experimental Scenario

To quantify the performance impact of the query resolution phase, we evaluate the contribution of the client-side caching mechanism. The Ermes Client Library natively implements a cache designed to prevent redundant resolution requests for recurrent queries. In this ablation study, we disable this feature and compare system performance against the baseline configuration. To simplify the experimental setup, we issue queries without explicit geolocation constraints, ensuring that they are invariably resolved by the Cloud node. We defer the detailed analysis of these results to the companion thesis, as they pertain primarily to implementation details rather than the data placement strategy, which constitutes the main focus of this work. Consequently, in the following, we limit our discussion to the interaction between the data placement policy (i.e., stateful nodes) and user caching.

Observed Results and Analysis

Figure 8.30 illustrates the user-perceived average latency and IQR after disabling the client-side caching of query resolutions. The results exhibit trends and curve shapes similar to those observed in the default configuration. The characteristic behaviors of the two consistency models, such as memory saturation on edge nodes for eventually consistent

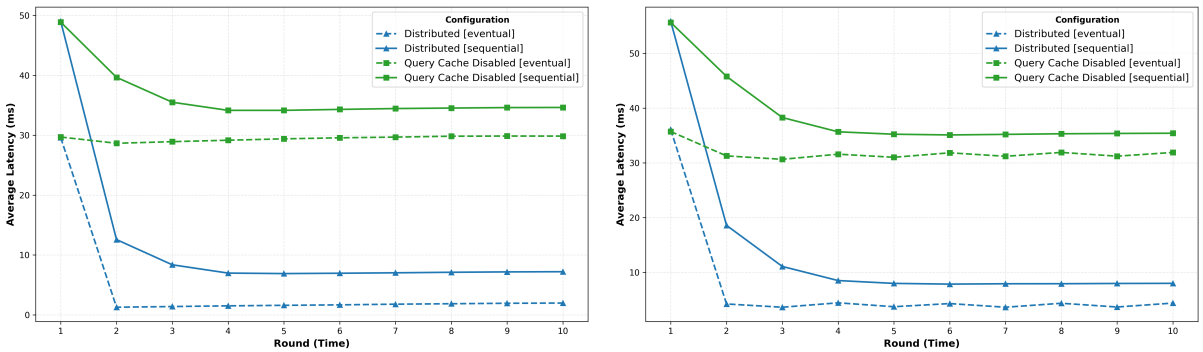
collections (Figure 8.30(a)) and increased latency due to write conflicts for sequentially consistent collections (Figure 8.30(b)), are still present. However, these effects are less pronounced, as they are partially masked by the significant latency overhead introduced by performing a remote query resolution for every invocation. This suggests that in this ablated configuration, the performance is dominated more by the query resolution overhead than by the data placement and distribution mechanisms.



(a) Eventual Consistency (b) Sequential Consistency

Figure 8.30: Scalability of user-perceived average latency in Ermes (client cache off).

Notably, the performance gap between the baseline and this cache-off scenario is comparable in magnitude to the gap observed between the baseline and the fully centralized (stateless) architecture. This finding underscores that client-side caching of metadata and decentralized data placement are synergistic and equally critical components; both must be active to achieve optimal system performance. Disabling either one results in a significant and comparable degradation in user-perceived latency.



(a) 100 collections (b) 1000 collections

Figure 8.31: Latency Time behaviour with 1024 users.

An analysis of the system's transient behavior, shown in Figure 8.31, further underscores

the critical role of the client-side cache. In the 100-collection scenario with 1024 users (Figure 8.31(a)), the latency for eventually consistent collections shows no improvement over time, even as our placement policy progressively moves data closer to the clients. The substantial, recurring overhead of performing a remote query resolution for every invocation completely masks the performance benefits of improved data locality. This result confirms that without effective metadata caching, the advantages of our dynamic data placement are nullified.

8.7. Robustness to User Mobility

Experimental Scenario

Having established the system's baseline behavior through the preceding ablation study, we now extend our analysis to evaluate robustness against user mobility. We model this scenario using a periodic, synchronized round-robin migration of all clients across the edge nodes, occurring every 5 user-rounds (equivalent to 75 seconds). This configuration represents a worst-case operational scenario: the simultaneous displacement of the entire user base triggers an instantaneous, system-wide invalidation of established data locality. Unlike asynchronous mobility patterns, which allow for incremental adaptation, this collective transition imposes a maximal stress test, subjecting the infrastructure to an abrupt spike in concurrent state migration demands. Consequently, rather than introducing a new experimental dimension, we select a representative operating point from the ablation study, fixed at 64 users and 1000 collections, to assess the system's capacity to recover from transient deviations from equilibrium.

We deliberately dimension the collection pool to be an order of magnitude larger than the user population. This design choice minimizes the probability that a collection required by a migrating client is pre-existing on the destination edge node due to incidental locality generated by previous occupants. By suppressing inter-user locality overlap as a confounding factor, we ensure that data availability at the destination is not an artifact of stochastic collisions. Consequently, the observed state-transfer spikes during the synchronized migration can be attributed primarily to the mobility stressor itself, allowing for a precise isolation of the migration overhead.

Observed Results and Analysis

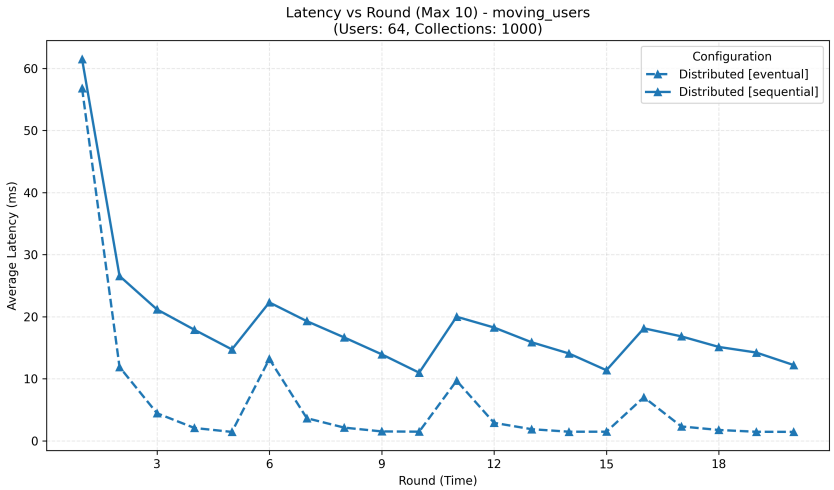


Figure 8.32: Latency Time behavior, 64 users and 1000 collections, with mobility perturbation.

Figure 8.32 illustrates the temporal evolution of average latency, sampled at 15-second intervals corresponding to user rounds, for both consistency levels under the default Ermes configuration. In both curves, we observe three distinct intervals where the latency gradient is positive, corresponding to the performance degradation induced by synchronized user migration events. Notably, the local maxima reached during these perturbation phases remain significantly lower than the global maximum associated with the initial cold-start phase. These results underscore how the trade-off between availability and consistency impacts not only the global average latency perceived by users but also the transient behavior of collections as they transition from an unbalanced to a balanced state.

We observe a significant disparity in convergence time between the two models. For eventually consistent collections (dotted line), the system requires only a single round to return to pre-perturbation latency levels. In contrast, sequentially consistent collections (solid line) necessitate approximately four rounds to re-establish equilibrium. This divergence is intrinsic to the distinct migration policies employed: the policy for eventual consistency triggers replication decisions upon every function invocation, allowing for rapid data propagation across the network hierarchy. Conversely, the policy for sequential consistency restricts migration decisions to discrete epoch boundaries (every 15 seconds), which operate asynchronously with respect to user rounds. Consequently, since the data must traverse multiple network tiers to reach the new optimal location, the latency exhibits a gradual, monotonic decrease over the four rounds following the invalidation of data locality.

8.8. Discussion of Outcomes and Benefits

In this section, we interpret the experimental results presented in this chapter, analyzing them in the context of our proposed model, placement policies, and research questions. Our discussion here focuses strictly on the empirical outcomes, deferring broader architectural conclusions and future work to Chapter 9. Overall, our experimental campaign confirms that our stateful, data-centric execution model performs as predicted, validating its benefits while also highlighting its inherent trade-offs.

The Limits of Centralized Optimality [RQ1]
The evaluation demonstrates that while variable reduction techniques significantly extend the tractable range of the mathematical model, centralized optimal orchestration remains fundamentally incompatible with the operational requirements of edge computing. Specifically, the analysis reveals that while the model can produce optimal baselines within 10 seconds for small clusters ($|N| = 10$), it encounters a "state obsolescence" threshold as workloads increase; instances with 100 nodes and high function counts require upwards of 10 minutes and 64 GB of memory to resolve. The findings confirm that the sequential consistency model is particularly prone to combinatorial explosions due to strict placement constraints. This identifies a clear threshold for "state obsolescence," where the computed optimal placement reflects a stale network state. Consequently, while the ILP model serves as a rigorous theoretical baseline for optimality, its computational overhead renders it unsuitable for production-level edge orchestration, thereby necessitating the adoption of low-latency decentralized heuristics.

Empirical Validation of Placement Policies Against Mathematical Model Predictions [RQ2]
A central goal of our evaluation was to empirically validate our distributed placement policy against the optimal placements predicted by the mathematical model. Our experiments demonstrate a strong correspondence between the equilibrium state reached by the system's implementation and the theoretical optimal configuration. This confirms that our distributed protocol, derived from the physical model, effectively converges to the predicted placement, thereby validating its correctness.

Convergence Dynamics and Transient Effects
Our experiments also highlight the critical role of convergence dynamics in overall system performance. During the initial epochs, the system operates in a transient, unbalanced state where replica placements are not yet optimal. This transient phase significantly

impacts user-perceived latency. We observed that the duration of this phase, measured in the number of epochs required to reach equilibrium, is a direct function of the network distance, in terms of hops, between a collection's initial location and its final, optimal placement. Once convergence is reached, latency values stabilize at a significantly lower level, reflecting the system's steady-state behavior.

Impact of Consistency Policies [RQ3]

The experimental results empirically validate the fundamental tradeoff between availability and semantic guarantees, identifying the consistency policy as a determinant factor in system performance and convergence dynamics. While sequential consistency enforces strict ordering, it incurs a significant performance penalty; the requirement for a single authoritative leader, combined with the dependency on periodic state dissemination to determine optimal placement, results in prolonged convergence times and higher overall latency. In contrast, eventual consistency prioritizes responsiveness, achieving rapid convergence and reduced latency by relaxing ordering constraints and permitting aggressive replication. However, our analysis reveals that this strategy incurs a distinct resource cost: the proliferation of replicas results in intensified memory pressure, particularly in scenarios characterized by a high density of collections.

Communication Overhead and State Propagation Costs

Regarding communication overhead, our experimental results indicate that update propagation and offloaded function invocations are the primary drivers of network traffic. In contrast, the gossip mechanism, essential for exchanging resource usage statistics and enforcing placement policies, contributes only a negligible fraction to the total message volume. We observe this trend consistently across diverse workloads and configurations, confirming that the cost of maintaining distributed state significantly outweighs the coordination overhead in the evaluated scenarios. Consequently, while the control traffic generated by our distributed migration policies remains minimal, the data demonstrates that replication intensity directly determines the overall network utilization.

Comparison with the Stateless Baseline [RQ4]

The ablation experiments comparing our proposed stateful architecture against a stateless baseline demonstrate consistent performance advantages. The distributed design yields significantly lower average latency and reduced interquartile ranges, a benefit that becomes particularly pronounced in scenarios involving repeated access to the same data artifacts. These performance gains are directly attributable to our architectural strategy of localizing state through caching and demand-driven replication, which effectively mit-

igates the latency overhead associated with centralized persistence and remote function invocations.

Secondary Effects: Caching, Scalability, and Mobility [RQ5–RQ6]
Our experiments also underscore the critical role of client-side caching and local persistence in mitigating response time and variability, particularly following system convergence. Furthermore, supplementary tests involving synchronized migration confirm the system's robustness under user mobility, revealing recovery dynamics that are strictly dependent on the adopted consistency model.

Regarding scalability, our results indicate that the proposed architecture scales effectively provided that sufficient memory resources are available. However, in configurations characterized by aggressive replication and a high volume of collections, we observed memory saturation at the edge nodes. This resource exhaustion necessitates the placement of collections on higher-tier nodes, inevitably resulting in increased latency. Consequently, we identify edge memory capacity as a physical constraint limiting the scalability of the real-world deployments.

Experimental and Architectural Limitations
While the experimental results provide strong empirical support for the proposed model and policies, we must contextualize these findings within the scope of the evaluated configurations. From an architectural perspective, the experiments confirm that aggressive replication strategies,while essential for achieving high responsiveness, inevitably increase memory pressure on edge nodes as the number of active collections expands. We interpret this behavior not as an unforeseen limitation, but as a manifestation of the fundamental trade-off between resource consumption and performance inherent to distributed systems, a relationship consistently corroborated by our experimental evidence.

Similarly, we designed the experimental methodology to isolate and stress-test specific system dynamics, including convergence behavior, replication overhead, and migration costs. By utilizing a controlled environment characterized by predefined network topologies, bounded latency ranges, and synthetic workloads, we ensure reproducibility and facilitate the precise attribution of observed effects to specific architectural mechanisms. While real-world deployments inevitably exhibit greater heterogeneity and variability, this methodology establishes a rigorous foundation for evaluating the core properties of the system. Consequently, the reported results represent the system's behavior under well-defined, analytically motivated conditions, rather than serving as absolute performance bounds.

9 | Conclusions and Future Work

In this thesis, we investigated the extension of stateful serverless computing to the network edge, a domain where latency, resource constraints, and data locality are primary concerns. Our work addresses a critical gap in existing FaaS platforms, which typically treat persistent state as an external dependency rather than a first-class citizen. This approach often forces developers to rely on centralized storage systems that are ill-suited for edge-native workloads. We therefore set out to determine whether a tighter integration of computation and state management could yield significant performance improvements while preserving the flexibility of the serverless model.

The central contribution of this work is a distributed, physics-inspired data placement policy that dynamically manages both data migration and replication. We grounded this policy in a formal mathematical model, formulated as an Integer Linear Programming (ILP) problem, that defines the optimal co-location of functions and data. This theoretical foundation serves a dual purpose: it provides a principled basis for deriving the decentralized decision rules of our placement algorithm, and it establishes an optimal baseline against which we validate our heuristic's performance.

To validate our architectural vision, we engineered and implemented Ermes, a complete FaaS platform. Our system is distinguished by several key features: (1) it treats persistent state as a first-class architectural component, not an external dependency; (2) it tightly integrates data placement with function scheduling to prioritize data locality; and (3) it supports configurable consistency guarantees while leveraging lightweight WebAssembly runtimes for efficient execution at the edge.

Our experimental evaluation empirically demonstrates that the proposed distributed placement mechanisms successfully approximate the optimal solutions derived from the ILP model, all while operating under realistic constraints and without centralized coordination. The results confirm that co-locating state with computation at the edge significantly reduces user-perceived latency. Concurrently, the evaluation quantifies the fundamental trade-offs inherent in this design: stronger consistency guarantees introduce coordination overhead that increases latency, while aggressive replication improves read performance

at the cost of higher memory consumption on resource-constrained edge nodes.

Beyond quantitative performance metrics, our evaluation also yielded insights into the system's dynamic behavior. We observed that convergence dynamics, transient states caused by workload shifts, and the impact of mechanisms like client-side caching all play a significant role in shaping end-to-end performance.

In conclusion, this thesis demonstrates that treating state as a first-class entity in serverless platforms is both feasible and highly beneficial, particularly for edge deployments where data locality is paramount. By explicitly reasoning about data placement, consistency, and execution locality, Ermes bridges a conceptual gap between traditional distributed storage systems and modern serverless execution models. Our work highlights the opportunities and trade-offs of this design space and provides a solid foundation for future research at the intersection of serverless computing, distributed systems, and edge infrastructure.

9.1. Directions for Future Work

While this thesis establishes a solid foundation for stateful serverless computing at the Edge, several promising avenues for future research remain. We identify the following key directions to extend our work:

- **Empirical Validation of Model–Physical Convergence:** We plan to empirically assess the convergence between the mathematical model and its physical counterpart using a simulated environment, decoupled from the Ermes implementation. This entails quantifying any discrepancy between the optimal solutions derived from the theoretical mathematical model and those obtained via the distributed solution induced by the physical representation. Additionally, we aim to compare the scalability of the two approaches with respect to network size, number of users, and number of collections.
- **Adaptive Placement Thresholds:** Our current data placement policy relies on static, globally configured thresholds. We plan to develop adaptive, per-collection thresholds to improve the framework's adaptability under heterogeneous and time-varying workloads. To this end, we will investigate machine learning techniques that allow the system to dynamically learn and adjust these thresholds based on observed access patterns.
- **Refined Resource Modeling:** The current placement heuristic primarily utilizes memory pressure as a proxy for overall node load. Future work should refine this ap-

proach by incorporating a more comprehensive resource model. We plan to monitor additional metrics, such as CPU utilization and network I/O, to provide the data migration and function scheduling policies with a more accurate view of resource contention on each node.

- **Fair Scheduling and Reliability at the Edge:** We plan to investigate the adoption of scheduling policies that balance the data locality introduced in this work with effective load-balancing mechanisms, aiming to mitigate the RAM saturation phenomena observed during our experiments. Additionally, we will explore strategies for introducing reliability in Edge computing environments, where resource constraints and fault tolerance are critical concerns.
- **GPU Workload Integration:** While the current implementation is limited to CPU-bound functions, extending Ermes to support GPU-accelerated workloads represents a significant research direction. We plan to investigate emerging WebAssembly runtimes with GPU capabilities or develop novel techniques to securely and efficiently offload computations to GPUs, enabling a new class of machine learning and signal processing applications at the Edge.

Addressing these challenges will enable the further refinement of the Ermes framework, paving the way for efficient and effective location-aware, stateful serverless computing at the Edge.

Bibliography

[1] Amazon Web Services. AWS IoT Greengrass. https://aws.amazon.com/greengrass/, 2026. Accessed: 2026-01-04.

[2] Amazon Web Services. AWS Lambda. https://aws.amazon.com/lambda/, 2026. Accessed: 2026-01-04.

[3] Amazon Web Services. AWS Lambda at Edge. https://aws.amazon.com/lambda/edge/, 2026. Accessed: 2026-01-04.

[4] Apache. Openwhisk. https://openwhisk.apache.org/, 2026. Accessed: 2026-01-04.

[5] D. Barcelona-Pons, M. Sánchez-Artigas, G. París, P. Sutra, and P. García-López. On the faas track: Building stateful distributed applications with serverless architectures. In Proceedings of the 20th International Middleware Conference, Middleware '19, page 41–54, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450370097. doi: 10.1145/3361525.3361535. URL https://doi.org/10.1145/3361525.3361535.

[6] D. Barcelona-Pons, P. Sutra, M. Sánchez-Artigas, G. París, and P. García-López. Stateful serverless computing with crucial. ACM Trans. Softw. Eng. Methodol., 31 (3), Mar. 2022. ISSN 1049-331X. doi: 10.1145/3490386. URL https://doi.org/10.1145/3490386.

[7] L. Baresi, D. Y. X. Hu, G. Quattrocchi, and L. Terracciano. Neptune: A comprehensive framework for managing serverless functions at the edge. ACM Trans. Auton. Adapt. Syst., 19(1), Feb. 2024. ISSN 1556-4665. doi: 10.1145/3634750. URL https://doi.org/10.1145/3634750.

[8] D. Breitgand. Lean OpenWhisk: Open Source FaaS for Edge Computing. https://medium.com/openwhisk/lean-openwhisk-open-source-faas-for-edge-computing-fb823c6bbb9b, July 2018. Accessed: 2026-01-04.

[9] D. Carrizales-Espinoza, D. D. Sanchez-Gallegos, J. Gonzalez-Compean, and J. Carretero. Structmesh: A storage framework for serverless computing continuum. Future Generation Computer Systems, 159:353–369, 2024. ISSN 0167-739X. doi: https://doi.org/10.1016/j.future.2024.05.033. URL https://www.sciencedirect.com/science/article/pii/S0167739X24002401.

[10] K. M. Chandy and L. Lamport. Distributed snapshots: determining global states of distributed systems. ACM Trans. Comput. Syst., 3(1):63–75, Feb. 1985. ISSN 0734-2071. doi: 10.1145/214451.214456.

[11] M. Ciavotta, D. Motterlini, M. Savi, and A. Tundo. Dfaas: Decentralized function-as-a-service for federated edge computing. In 2021 IEEE 10th International Conference on Cloud Networking (CloudNet), pages 1–4, 2021. doi: 10.1109/CloudNet53349.2021.9657141.

[12] C. Cicconetti, M. Conti, and A. Passarella. A decentralized framework for serverless edge computing in the internet of things. IEEE Transactions on Network and Service Management, 18(2):2166–2180, 2021. doi: 10.1109/TNSM.2020.3023305.

[13] C. Cicconetti, E. Carlini, R. Hetzel, R. Mortier, A. Paradell, and M. Sauer. Edgeless: A software architecture for stateful faas at the edge. In Proceedings of the 33rd International Symposium on High-Performance Parallel and Distributed Computing, HPDC '24, page 393–396, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400704130. doi: 10.1145/3625549.3658817. URL https://doi.org/10.1145/3625549.3658817.

[14] Cloudflare. Cloudflare Workers. https://workers.cloudflare.com/, 2026. Accessed: 2026-01-04.

[15] A. Davoudian, L. Chen, and M. Liu. A survey on nosql stores. ACM Comput. Surv., 51(2), Apr. 2018. ISSN 0360-0300. doi: 10.1145/3158661. URL https://doi.org/10.1145/3158661.

[16] G. De Palma, S. Giallorenzo, J. Mauro, M. Trentin, and G. Zavattaro. Funless: Functions-as-a-service for private edge cloud systems. In 2024 IEEE International Conference on Web Services (ICWS), pages 961–967, 2024. doi: 10.1109/ICWS62655.2024.00114.

[17] M. Diogo, B. Cabral, and J. Bernardino. Consistency models of nosql databases. Future Internet, 11(2):43, 2019.

[18] S. Dustdar, V. C. Pujol, and P. K. Donta. On distributed computing continuum

systems. IEEE Transactions on Knowledge and Data Engineering, 35(4):4092–4105, 2023. doi: 10.1109/TKDE.2022.3142856.

[19] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. Commun. ACM, 19(11):624–633, Nov. 1976. ISSN 0001-0782. doi: 10.1145/360363.360369.

[20] P. K. Gadepalli, S. McBride, G. Peach, L. Cherkasova, and G. Parmer. Sledge: a serverless-first, light-weight wasm runtime for the edge. In Proceedings of the 21st International Middleware Conference, Middleware '20, page 265–279, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450381536. doi: 10.1145/3423211.3425680. URL https://doi.org/10.1145/3423211.3425680.

[21] Google. The Go Programming Language. https://go.dev/, 2026. Accessed: 2026-01-04.

[22] Google. Google Cloud Functions. https://cloud.google.com/functions, 2026. Accessed: 2026-01-04.

[23] Google. Goroutines explained. https://go.dev/tour/concurrency/1, 2026. Accessed: 2026-01-04.

[24] I. Griva, S. G. Nash, and A. Sofer. Linear and nonlinear optimization 2nd edition. SIAM, 2008.

[25] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien. Bringing the web up to speed with webassembly. SIGPLAN Not., 52(6):185–200, June 2017. ISSN 0362-1340. doi: 10.1145/3140587.3062363.

[26] J. M. Hellerstein, J. Faleiro, J. E. Gonzalez, J. Schleier-Smith, V. Sreekanti, A. Tumanov, and C. Wu. Serverless computing: One step forward, two steps back. arXiv preprint arXiv:1812.03651, 2018.

[27] C.-H. Hong and B. Varghese. Resource management in fog/edge computing: A survey on architectures, infrastructure, and algorithms. ACM Comput. Surv., 52(5), Sept. 2019. ISSN 0360-0300. doi: 10.1145/3326066. URL https://doi.org/10.1145/3326066.

[28] N. Jain and S. Choudhary. Overview of virtualization in cloud computing. In 2016 Symposium on Colossal Data Analysis and Networking (CDAN), pages 1–4, 2016. doi: 10.1109/CDAN.2016.7570950.

[29] E. Jonas, J. Schleier-Smith, V. Sreekanti, C. Tsai, A. Khandelwal, Q. Pu, V. Shankar,

J. Carreira, K. Krauth, N. J. Yadwadkar, J. E. Gonzalez, R. A. Popa, I. Stoica, and D. A. Patterson. Cloud programming simplified: A berkeley view on serverless computing. CoRR, abs/1902.03383, 2019. URL http://arxiv.org/abs/1902.03383.

[30] W. Z. Khan, E. Ahmed, S. Hakak, I. Yaqoob, and A. Ahmed. Edge computing: A survey. Future Generation Computer Systems, 97:219–235, 2019. ISSN 0167-739X. doi: https://doi.org/10.1016/j.future.2019.02.050.

[31] Y. Li, Y. Lin, Y. Wang, K. Ye, and C. Xu. Serverless computing: State-of-the-art, challenges and opportunities. IEEE Transactions on Services Computing, 16(2): 1522–1539, 2023. doi: 10.1109/TSC.2022.3166553.

[32] M. Liu, H. Shen, Y. Zhang, H. Mei, and Y. Ma. Webassembly for container runtime: Are we there yet? ACM Trans. Softw. Eng. Methodol., 34(6), July 2025. ISSN 1049-331X. doi: 10.1145/3712197.

[33] R. V. Lopes and D. Menascé. A taxonomy of job scheduling on distributed computing systems. IEEE Transactions on Parallel and Distributed Systems, 2016. doi: 10.1109/TPDS.2016.2537821.

[34] R. Mahmud, K. Ramamohanarao, and R. Buyya. Application management in fog computing environments: A taxonomy, review and future directions. ACM Comput. Surv., 53(4), July 2020. ISSN 0360-0300. doi: 10.1145/3403955. URL https://doi.org/10.1145/3403955.

[35] Y. Mao, C. You, J. Zhang, K. Huang, and K. B. Letaief. A survey on mobile edge computing: The communication perspective. IEEE Communications Surveys Tutorials, 19(4):2322–2358, 2017. doi: 10.1109/COMST.2017.2745201.

[36] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard Version 4.0, June 2021. URL https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf.

[37] Microsoft. Microsoft Azure Durable Functions. https://learn.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-overview, 2026. Accessed: 2026-01-04.

[38] Microsoft. Microsoft Azure Functions. https://azure.microsoft.com/en-us/products/functions, 2026. Accessed: 2026-01-04.

[39] Microsoft Developers. Azure Durable Tasks. https://aws.amazon.com/lambda/, 2026. Accessed: 2026-01-04.

[40] M. Nardelli and G. Russo Russo. Function offloading and data migration for stateful serverless edge computing. In Proceedings of the 15th ACM/SPEC International Conference on Performance Engineering, ICPE '24, page 247–257, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400704444. doi: 10.1145/3629526.3649293. URL https://doi.org/10.1145/3629526.3649293.

[41] S. Nastic, T. Rausch, O. Scekic, S. Dustdar, M. Gusev, B. Koteska, M. Kostoska, B. Jakimovski, S. Ristov, and R. Prodan. A serverless real-time data analytics platform for edge computing. IEEE Internet Computing, 21(4):64–71, 2017. doi: 10.1109/MIC.2017.2911430.

[42] netbeez. Understanding network latency. URL https://netbeez.net/blog/network-latency/.

[43] OpenFaas developers. Openfaas. https://www.openfaas.com/, 2026. Accessed: 2026-01-04.

[44] G. Orsenigo and L. Tosetti. Ermes 2.0. Master's thesis, Politecnico di Milano, 2026.

[45] T. Pfandzelter and D. Bermbach. tinyfaas: A lightweight faas platform for edge environments. In 2020 IEEE International Conference on Fog Computing (ICFC), pages 17–24, 2020. doi: 10.1109/ICFC49376.2020.00011.

[46] T. Pfandzelter and D. Bermbach. Enoki: Stateful distributed faas from edge to cloud. In Proceedings of the 2nd International Workshop on Middleware for the Edge, MiddleWEdge '23, page 19–24, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400704512. doi: 10.1145/3630180.3631203. URL https://doi.org/10.1145/3630180.3631203.

[47] T. Pfandzelter, N. Japke, T. Schirmer, J. Hasenburg, and D. Bermbach. Managing data replication and distribution in the fog with fred. Software: Practice and Experience, 53(10):1958–1981, 2023.

[48] P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. Seltzer. Network-aware operator placement for stream-processing systems. In 22nd International Conference on Data Engineering (ICDE'06), pages 49–49, 2006. doi: 10.1109/ICDE.2006.105.

[49] A. M. Potdar, N. D G, S. Kengond, and M. M. Mulla. Performance evaluation of docker container and virtual machine. Procedia Computer Science, 171:1419–1428, 2020. ISSN 1877-0509. doi: https://doi.org/10.1016/j.procs.2020.04.152. Third International Conference on Computing and Network Communications (CoCoNet'19).

[50] A. Pérez, G. Moltó, M. Caballer, and A. Calatrava. Serverless computing for container-based architectures. Future Generation Computer Systems, 83:50–59, 2018. ISSN 0167-739X. doi: https://doi.org/10.1016/j.future.2018.01.022.

[51] F. Ramalho and A. Neto. Virtualization at the network edge: A performance comparison. In 2016 IEEE 17th International Symposium on A World of Wireless, Mobile and Multimedia Networks (WoWMoM), pages 1–6, 2016. doi: 10.1109/WoWMoM.2016.7523584.

[52] M. Raynal. About logical clocks for distributed systems. SIGOPS Oper. Syst. Rev., 26(1):41–48, Jan. 1992. ISSN 0163-5980. doi: 10.1145/130704.130708.

[53] G. R. Russo, T. Mannucci, V. Cardellini, and F. L. Presti. Serverledge: Decentralized function-as-a-service for the edge-cloud continuum. In 2023 IEEE International Conference on Pervasive Computing and Communications (PerCom), pages 131–140, 2023. doi: 10.1109/PERCOM56429.2023.10099372.

[54] R. S. Sandhu. Role-based access control. Advances in Computers. Elsevier, 1998. doi: https://doi.org/10.1016/S0065-2458(08)60206-5. URL `https://www.sciencedirect.com/science/article/pii/S0065245808602065`.

[55] S. Sanfilippo. Redis. `https://redis.io/`, 2026. Accessed: 2026-01-04.

[56] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. Research Report RR-7506, Inria – Centre Paris-Rocquencourt ; INRIA, Jan. 2011. URL `https://inria.hal.science/inria-00555588`.

[57] S. Shillaker and P. Pietzuch. Faasm: Lightweight isolation for efficient stateful serverless computing. In 2020 USENIX Annual Technical Conference (USENIX ATC 20), pages 419–433. USENIX Association, July 2020. ISBN 978-1-939133-14-4. URL `https://www.usenix.org/conference/atc20/presentation/shillaker`.

[58] C. P. Smith, A. Jindal, M. Chadha, M. Gerndt, and S. Benedict. Fado: Faas functions and data orchestrator for multiple serverless edge-cloud clusters. In 2022 IEEE 6th International Conference on Fog and Edge Computing (ICFEC), pages 17–25, 2022. doi: 10.1109/ICFEC54809.2022.00010.

[59] S. Vahabi, F. Righetti, C. Vallati, and N. Tonellotto. Energy-efficient resource management for real-time applications in faas edge computing platforms. In Proceedings of the IEEE/ACM 16th International Conference on Utility and Cloud Computing, UCC '23, New York, NY, USA, 2024. Association for Computing Machinery. ISBN

9798400702341. doi: 10.1145/3603166.3632240. URL https://doi.org/10.1145/3603166.3632240.

[60] K. Varda. Introducing Workers Durable Objects. https://blog.cloudflare.com/introducing-workers-durable-objects/, Sept. 2020. Accessed: 2026-01-04.

[61] P. Viotti and M. Vukolić. Consistency in non-transactional distributed storage systems. ACM Comput. Surv., 49(1), June 2016. ISSN 0360-0300. doi: 10.1145/2926965. URL https://doi.org/10.1145/2926965.

[62] WASI developers. Webassembly system interface. https://github.com/WebAssembly/WASI, 2026. Accessed: 2026-01-04.

A | Ermes Framework Details

A.1. Metadata Schemas

Field	Description
`name`	Name of the collection
`ownerID`	Identifier of the collection owner
`created_at`	Collection creation timestamp
`groups`	List of groups authorized to access the collection
`consistency`	Consistency level (Strong Eventual or Sequential)

Table A.1: Collection Metadata fields.

Field	Description
`name`	Human-readable function identifier
`version`	Function version number
`deployed_on`	Cloud regions where the function is deployed
`query_templates`	Query templates the function is permitted to use
`is_auth_function`	Indicates whether the function performs authentication
`sequential_write_access`	Allows writes on sequentially consistent collections

Table A.2: Function Metadata fields.

Parameter	Description
`User Groups`	List of group memberships extracted from the *Group Token*, used to filter query results and enforce access control
`Geolocation`	Optional hint indicating the domain where the collection metadata is anchored, used to optimize query routing
`Metadata Query Template IDs`	Identifiers of the *Query Templates* associated with the invoked function
`Metadata Query Template Parameters`	Runtime values used to complete the specified query templates and resolve data dependencies

Table A.3: Metadata Lookup input parameters

B | Ermes Framework Evaluation Details

B.1. Use-case-driven experiments

Setting name	Value	Description
WASM_POOL_SIZE	10	The maximum number of concurrent WASM function instances per node.
LRU size T3	10000	The maximum number of items in the LRU cache for Tier 3 nodes.
LRU size T2	5000	The maximum number of items in the LRU cache for Tier 2 nodes.
LRU size T1	2000	The maximum number of items in the LRU cache for Tier 1 nodes.
LRU safeLevel	0.8	The cache fill percentage that triggers collection offloading during the Decision state.
redis_maxmemory T3	4gb	The maximum memory allocated to the Redis instance on Tier 3 nodes.
redis_maxmemory T2	512mb	The maximum memory allocated to the Redis instance on Tier 2 nodes.
redis_maxmemory T1	256mb	The maximum memory allocated to the Redis instance on Tier 1 nodes.
leaderThreshold	700	The decision threshold for migrating a leader replica in the physical model.
followerThreshold	150	The decision threshold for migrating a follower replica in the physical model.
viewSize	5	The number of recent epochs considered in placement decisions.
idleTimeoutMS	15000	The duration of the Idle state in milliseconds.
gravityConstant	9.81	The value of the gravity constant (G) used in the physical model.
knownCollectionsDumpInterval	7s	-

Table B.1: Ermes settings for Use-case-driven experiments.

B.2. Ablation Study

Setting name	Value	Description
WASM_POOL_SIZE_T3	1000	The maximum number of concurrent WASM function instances on the cloud node.
WASM_POOL_SIZE_T2	500	The maximum number of concurrent WASM function instances per fog node.
WASM_POOL_SIZE_T1	50	The maximum number of concurrent WASM function instances per edge node.
LRU size T3	100000000000	The maximum number of items in the LRU cache for Tier 3 nodes.
LRU size T2	1000000	The maximum number of items in the LRU cache for Tier 2 nodes.
LRU size T1	600000	The maximum number of items in the LRU cache for Tier 1 nodes.
LRU safeLevel	0.8	The cache utilization threshold that triggers collection offloading during the Decision phase.

Table B.2: Ermes settings for Ablation Study experiments.

List of Figures

List of Tables

List of Symbols

Variable	Description	SI unit
$\boldsymbol{\rho}_{\mathrm{f}}$	fluid density	$\frac{kg}{m^3}$
$\boldsymbol{V}$	volume	m^3
$\vec{\boldsymbol{f}}_{\boldsymbol{E}}$	elastic force	N
$\vec{\boldsymbol{f}}_{\boldsymbol{B}}$	buoyant force	N
$\boldsymbol{E}_{\boldsymbol{elastic}}$	elastic potential energy	J
$\boldsymbol{E}_{\boldsymbol{hydrostatic}}$	hydrostatic potential energy	J

Acknowledgements

First of all, we would like to thank our advisor Prof. Alessandro Margara and our co-advisors Arianna Dragoni and Dario D'Abate. Their constant support and advice has helped us producing a complete and formal work. Moreover, we would like to thank the authors of our parallel work, Giacomo Orsenigo and Luca Tosetti, with whom we have shared the joys and hurdles of building Ermes.

Our special thanks are also extended to Politecnico di Milano, with all its professors, our colleagues and friends. This academic journey has been an incredibly valuable experience, both professionally and personally. During these difficult and long 5 years we had the opportunity to deepen our knowledge, and exchange ideas with brilliant people, with whom we have created strong bonds and connections.

Finally, and most importantly, a heartfelt thanks goes to our families. Their love and support has helped us through all these years in maintaining our strength and will to graduate even in our most difficult moments. Without them, none of this would have been possible.