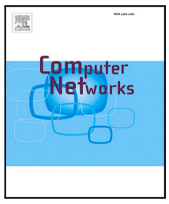
Computer Networks 262 (2025) 111158

Contents lists available at ScienceDirect

Computer Networks

journal homepage: www.elsevier.com/locate/comnet



Enabling efficient collection and usage of network performance metrics at the edge

Antonio Calagna [a,*], Stefano Ravera [a], Carla Fabiana Chiasserini [a,b,c]

[a] Politecnico di Torino, Turin, Italy
[b] CNIT, Parma, Italy
[c] Chalmers University of Technology, Göteborg, Sweden

ARTICLE INFO

Keywords:
Edge computing
Microservices
Data consistency
Data availability

ABSTRACT

Microservices (MSs)-based architectures have become the de facto standard for designing and implementing edge computing applications. In particular, by leveraging Network Performance Metrics (NPMs) coming from the Radio Access Network (RAN) and sharing context-related information, AI-driven MSs have demonstrated to be highly effective in optimizing RAN performance. In this context, this work addresses the critical challenge of ensuring efficient data sharing and consistency by proposing a holistic platform that regulates the collection and usage of NPMs. We first introduce two reference platform architectures and detail their implementation using popular, off-the-shelf database solutions. Then, to evaluate and compare such architectures and their implementation, we develop PACE, a highly configurable, scalable, MS-based emulation framework of producers and consumers of NPMs, capable of realistically reproducing a broad range of interaction patterns and load dynamics. Using PACE on our cloud computing testbed, we conduct a thorough characterization of various NPM platform architectures and implementations under a spectrum of realistic edge traffic scenarios, from loosely coupled control loops to latency- and mission- critical use cases. Our results reveal fundamental trade-offs in stability, availability, scalability, resource usage, and energy footprint, demonstrating how PACE effectively enables the identification of suitable platform solutions depending on the reference edge scenario and the required levels of reliability and data consistency.

1. Introduction

Edge computing has emerged as a key paradigm for real-time data processing in proximity to data sources; in parallel, MS-based architectures have become the dominant design approach for edge applications, as they offer enhanced resiliency, isolation, and scalability features. In this context, next-generation networks are evolving toward RAN disaggregation and virtualization, as well as deep integration with data-driven control loops to attain unprecedented flexibility, resiliency, and reconfigurability. Such control loops can be actuated via intelligent applications, hosted as MSs at the edge, that leverage NPMs coming from the RAN to perform inference and forecasting, or to compute policies for network performance optimization. In fact, such NPMs provide valuable insights on the status of the network infrastructure (e.g., number of users, load, throughput, resource utilization), as well as additional context information from sources outside of the RAN. Relevant examples of MSs using NPMs to accomplish their tasks include self-driving vehicles controllers and AI-driven virtual network functions (VNFs) for traffic steering, handover management, slicing, and scheduling of network resources. Despite our work focuses on NPMs, our findings are broadly applicable and can be extended to any kind of information that could be relevant to share among MSs.

Despite the advantages brought by MS architectures, some *key issues related to their inherently decentralized data management* still need to be addressed, including coordination, data sharing, security, and consistency guarantees [1–3]. These aspects are particularly critical in distributed MSs scenarios or when multiple entities require concurrent access to shared data, e.g., NPMs coming from the RAN, and, importantly, they cannot be solved through traditional database solutions. Thus, in this work, we aim to shed light on the above issues, emphasizing that overcoming them is vital for enabling reliable, efficient, and scalable edge computing solutions. To effectively and efficiently regulate NPMs collection and usage, i.e., to deliver these metrics from the RAN to MSs and from one MS to another, we envision a holistic NPM platform, running at the edge, and acting as an abstraction layer between the edge applications and a database where data is stored. The platform not only coordinates cluster-wide NPMs consumption

* Corresponding author.
E-mail address: antonio.calagna@polito.it (A. Calagna).

https://doi.org/10.1016/j.comnet.2025.111158
Received 30 December 2024; Received in revised form 30 January 2025; Accepted 20 February 2025
Available online 2 March 2025

and retention but also acts as a common state repository that enables decoupling MSs from their state by storing it in the database. This is particularly relevant, but not limited to, stateful MSs (e.g., those used in forecasting, beam tracking, and mobility management), which need to maintain a history of context-based data to accomplish their tasks. By storing such data externally through the NPM platform, these MSs can be seamlessly scaled or migrated across edge nodes while ensuring service continuity. Also, being holistic, the platform can inherently enhance the security of microservices architectures by enforcing, e.g., encryption and authentication, by design [4], with no impact on the considerations drawn in our study.

Defining an NPM platform, however, poses several *technical challenges*, namely, *(i)* ensuring resilience to edge node failures and network partitions; *(ii)* enabling seamless scalability with the amount of data being retained, the number of MSs, and varying traffic load patterns; *(iii)* providing strong data consistency guarantees without excessively compromising availability or becoming a bottleneck for system performance; and *(iv)* thoroughly characterizing potential implementations to identify the one that best meets the strict requirements of latency-sensitive and mission-critical edge applications while minimizing resource consumption and energy footprint. To tackle these challenges, our work presents a comprehensive analysis of the trade-offs associated with different NPM platform architectures, focusing on essential real-world aspects of edge MS architectures, such as scalability, availability, and consistency. Leveraging our testbed based on a real-world cloud computing architecture, we deploy and evaluate these architectures under various implementations using popular, off-the-shelf database solutions. To do so, we develop PACE [5], a finely tunable emulation framework of producers and consumers of NPMs, which allows us to rigorously evaluate each solution under increasing complexity and diverse traffic scenarios. Our findings reveal several trade-offs in latency, data consistency, scalability, resource usage, and energy efficiency, providing crucial insights for determining the suitability of each approach to a specific edge scenario.

To summarize, our main contributions are as follows.

- We present two fundamental platform architectures for the collection and usage of performance metrics at the network edge, namely, a centralized and a distributed one, and highlight their respective advantages and disadvantages, encompassing all the relevant real-world aspects of MS-based architectures;
- We detail the implementation of these architectures using popular, off-the-shelf database solutions, e.g., *etcd* [6] and *Redis* [7], focusing on the different trade-offs between availability and consistency they can offer. Further, we specify the most relevant data types and APIs to use for the implementation of both architectures;
- We develop PACE,[1] a scalable Producer And Consumer Emulator that can model a wide range of interaction patterns and load dynamics, and we describe how to seamlessly integrate its components and information flow with our NPM platform implementation;
- Leveraging our testbed based on a real-world cloud computing architecture, we configure PACE to emulate diverse, realistic edge scenarios and perform a thorough experimental analysis of the NPM platform architectures and implementations we proposed under varying operational conditions;
- Based on our findings, we highlight the major learned lessons about the trade-offs inherent in different architectures and implementations and emphasize the importance of a careful configuration as well as the scalability and suitability of each solution for specific edge scenarios and applications.

The rest of the paper is organized as follows. Before discussing the NPM platform architectures and implementations in Section 3, we review relevant related works and emphasize the uniqueness of our study in Section 2. Then, we introduce our PACE framework in Section 4, which we configure to replicate realistic traffic scenarios as outlined in Section 5. Finally, we conduct a comprehensive performance evaluation in Section 6 and draw our conclusions in Section 7.

[1] https://github.com/antoniocalagna98/PACE

2. Related work

There exists a considerable amount of literature regarding MSs at the network edge, with some recent surveys available in [8–10]. Most of such prior art focuses on ensuring proximity of time- and mission-critical tasks with mobile end devices. For example, [11–14] analyze AI-driven and UAV-assisted tasks at the network edge for varying use case scenarios, [15–17] delve into task offloading and resource allocation within dense RANs, and [18] presents a smart traffic monitoring system that analyzes real-time video sources to perform monitoring tasks, congestion detection, and speed measurement. Further, many works highlight the importance of uninterrupted data exchange with the RAN to ensure reliable system control. Among these studies, [19, 20] investigates vehicular networks supported by edge computing, [21] elaborates on real-time surveillance applications in intelligent transportation systems, and [22] leverages real-world data traffic from a dense urban cellular network to define a cell-splitting-based radio resource management scheme. Interestingly, [23] remarks the crucial role of efficient data collection and processing architectures at the edge for health monitoring systems. [24–26], instead, demonstrate the effectiveness of AI-driven methodologies to harness RAN metrics in near-real-time to compute optimal policies for, e.g., traffic steering as well as scheduling and slicing of network resources under diverse traffic and channel conditions.

In spite of the numerous works in the field, however, few studies have tackled the challenges of effective data management, consistency, and sharing in MSs architectures. This is a significant gap since data sharing and consistency is of fundamental importance in edge computing architectures. Among the existing works, [27] introduces the most relevant database solutions for sensor networks, content delivery networks, and autonomous vehicles, while [28] proposes adaptive data placement in distributed Key-Value (KV) stores to reduce end-to-end latency. From an architectural perspective, a growing concern regarding the need to rethink traditional database architectures in the MSs context is raised in [2,29]. Specifically, it is observed that the inherently decentralized data management of MSs architectures poses significant challenges for coordination, as state dependencies and consistency issues are often overlooked. While most architectures feature a database-per-MS approach to improve performance and enforce logical boundaries, a non-negligible amount of applications require strong consistency guarantees over the shared information they access, which traditional loosely-coupled database systems cannot provide. This calls for a holistic central data governance paradigm, such as using a single, scalable and distributed database system to manage the states of all MSs.

Supporting the above vision, [30] proposes the definition of a common state repository to defer and share the MSs states, thus allowing for the refactoring of MSs into entirely stateless ones, in compliance with the requirements of 5G-and-beyond networks [31] and for the purpose of increased availability, load balancing, and reliability through redundancy. Similarly, the O-RAN alliance [32], which promotes enhanced flexibility via RAN disaggregation and virtualization, introduces the Shared Data Layer (SDL), a data access platform acting as an abstraction layer between AI-driven applications and a backend database where data is stored and shared. This approach effectively regulates data production and consumption between the RAN and such applications, as well as between one application and another [33]. We remark that this approach also facilitates the mobility of MSs across edge nodes, as it permits to effectively minimize the service disruption due to the migration process while jointly guaranteeing service continuity, and maintaining proximity to mobile end users [34].

Novelty of our work. As highlighted above, decentralized data management in MSs architectures poses significant challenges related to coordination, state dependencies, and data consistency, calling for a holistic data governance paradigm. Also, to enable an effective data production and consumption between the RAN and edge MSs, as well as

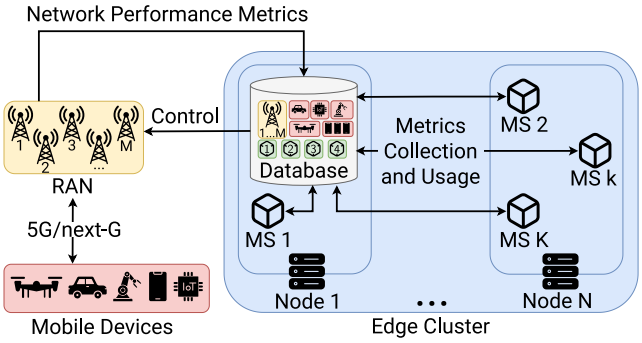


Fig. 1. Centralized NPM platform architecture, featuring only one database instance.

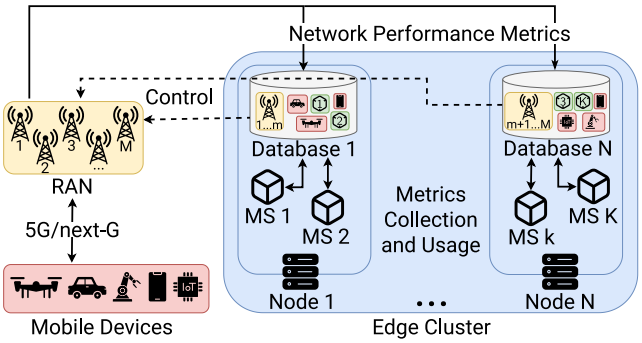


Fig. 2. Distributed NPM platform architecture, featuring multiple database instances across the computing cluster.

cooperative information sharing among independent MSs, a robust data access platform has yet to be developed. To address these challenges, we envision a holistic NPM platform at the edge that effectively and efficiently collects metrics coming from the RAN and exposes them to MSs for their optimization and control tasks. This platform also retains and shares MSs states, enabling the refactoring of MSs into stateless ones and ensuring that critical data consistency guarantees are always fulfilled. Our work is the first to thoroughly analyze the trade-offs of various NPM platform architectures, accounting for all the relevant real-world aspects of edge MSs architectures, such as scalability, availability, and consistency. Using our testbed setup, we deploy and evaluate these architectures under implementations leveraging different off-the-shelf database solutions, identifying if and to what extent these implementations are compatible with varying NPMs traffic scenarios. To do so, we propose PACE, a novel emulator of NPMs producers and consumers, all implemented as MSs, which captures a wide range of interaction patterns and computing load dynamics. In our performance evaluation, we use PACE to experimentally compare varying NPM platform architectures and implementations and to characterize their trade-offs in performance, resource usage, and energy footprint. Specifically, by utilizing a real cloud computing architecture and accounting for both active and idle energy consumption, we show that the trade-off between consistency and availability is further influenced by factors such as resilience to node failures and energy efficiency.

3. NPMs collection and usage: Platform architectures

To design a platform for efficient collection and usage of NPMs, either a centralized or a distributed architecture can be defined, depending on the platform database being used. In this section, we first introduce our reference scenario (Section 3.1) and then use it to describe the two types of architecture, along with the advantages they offer and the hurdles they exhibit. Finally, we present the platform implementations we developed, along with the off-the-shelf tools we used (Section 3.2).

3.1. Reference scenario and platform architectures

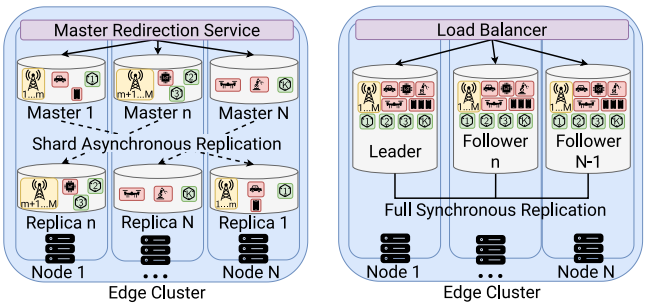
For concreteness, we focus on a scenario including a RAN with M gNBs and an edge cluster of N nodes, hosting a total of K MSs. The RAN periodically produces NPMs expressing the most updated state of the network, which are then used by the edge MSs to perform inference or compute policies aimed at optimizing the network performance. To deliver and retain such metrics, the NPM platform integrates a database whose architecture can be centralized or distributed.

Centralized architecture. As depicted in Fig. 1, in this case the NPMs platform comprises only one database instance running in a cluster node and storing (i) the NPMs from the RAN; (ii) context-related information, e.g., information on the mobile devices connected to the network; (iii) the internal state of each stateful MSs, representing, e.g., the history of context-based data that is needed as input to the MS; and (iv) the control policy generated by the MSs as a result of their inference process. The database thus represents a central, cluster-wide point of aggregation and access for all relevant information related to the network state.

As this solution features a single instance with no data replication, it is straightforward to implement and requires low computational and disk resources. However, it also introduces a single point of failure: in the case of node failure or temporary network partitions, data may be lost and all processes of NPM and control exchange may get blocked indefinitely. Additionally, such an architecture poses scalability limitations, as all data access requests are directed toward a single node, which may become overloaded and, hence, lead to increased latency and ineffective control policies.

Distributed architecture. Fig. 2 depicts a distributed architecture consisting of several database instances, one per edge node. By distributing the state over multiple edge nodes, the system is now resilient to node failures and can handle an increasing traffic load by distributing (hence parallelizing) data access requests across the multiple database instances. On the downside, additional mechanisms are needed to manage state replication and synchronization across the different instances, introducing a complexity that may impact resource usage and communication latency. In this case, we impose that network *partition tolerance* must always be ensured, as it is of paramount importance that the edge system and its time-critical applications remain resilient to node failures. It follows that, in accordance with the CAP theorem [35] stating that in distributed databases only two properties among *consistency*, *availability*, and *partition tolerance* can be guaranteed at any time,[2] consistency should be traded off with availability. When choosing consistency over availability, the system will return an error or a timeout whenever it cannot ensure that data is up-to-date. Conversely, when choosing availability over consistency, the system will always process a query and return the most recent available version of the data, even if outdated. Prioritizing one property over the other determines different ways in which data is physically organized and synchronized. We thoroughly analyze these aspects in the following, accounting for popular off-the-shelf technologies. Then, leveraging on our novel emulator, we experimentally characterize the impact of the trade-off between availability and consistency on the NPM platform performance.

[2] Consistency is the property ensuring that all replicas maintain the same view of the data. Availability is the property guaranteeing that every request eventually is fulfilled. Partition tolerance is the property assuring that the database continues to operate despite an arbitrary number of messages being dropped or delayed by the network.



(a) Data Sharding (b) Full Replication

Fig. 3. Distributed NPM platform implementation: (a) data sharding with asynchronous replication and (b) full synchronous replication.

To better illustrate this trade-off, we provide two examples of distributed database implementation: one prioritizing availability over consistency (Fig. 3(a)), and one focusing on consistency over availability (Fig. 3(b)). Fig. 3(a) shows a distributed database architecture based on data sharding and on the conceptual distinction between masters and masters' replicas. This architecture segments the data into *shards*, each managed by a specific *master* instance. A master redirection service allows clients to transparently access the appropriate shard, while replicas' only role is to mirror data from each master and supersede as new masters in case of failures. Each master asynchronously replicates its data to its replica, which is located on a different node for fault-tolerance. This design thus prioritizes high read/write performance (hence, availability) since replication does not need to be completed before responding to client requests. However, it may impair data consistency: if a master fails before propagating recent changes to its replica, those changes will be lost and the client is not notified of such loss. Fig. 3(b) shows instead a distributed database architecture based on full synchronous replication among all database instances, utilizing a leader-follower consensus model. Here, each instance holds an exact copy of the dataset, hence client requests can be equally distributed across the different instances through a load balancer. A quorum-based approach can then be adopted to ensure consistency, i.e., that data updates are acknowledged by the majority of instances before clients' requests are addressed and every read operation always returns the most up-to-date value. However, the associated communication overhead reduces availability, particularly in terms of response latency, as the response to any request involves the majority of the nodes.

3.2. NPM platform implementation

We now describe how to implement a centralized and a distributed architecture of an NPM platform using popular, off-the-shelf database technologies. Specifically, we select *etcd* [6] for prioritizing consistency over availability and *Redis* [7] for prioritizing availability over consistency. We remark that both databases are widely used in the literature as well as in practical applications [36–39].

Database technologies. *Etcd* is a highly reliable, distributed key–value (KV) store designed for managing the most critical data of a distributed system. By leveraging the Raft [40] consensus algorithm, etcd enforces both strong data consistency and tolerance to network partitions and machine failure at the cost of reduced availability. The difference between implementing centralized and distributed architectures with etcd lies on the number of members in the etcd environment: a single-member configuration disables Raft consensus, representing a centralized setup, while a distributed setup consists of N etcd members, each corresponding to an edge node in the cluster.

To guarantee high reliability, etcd stores entries in a multi-version persistent KV format, retaining prior versions of KV pairs each time a value is updated. As a result, etcd keeps an exact history of its keyspace, which should be periodically *compacted* to avoid performance degradation and eventual storage space exhaustion. Since compacting old revisions internally fragments etcd by leaving gaps in the database, it is also necessary to release from time to time this storage space back to the file system through a *defragmentation* process. In the following, we refer to the combination of the compaction and defragmentation processes as a *maintenance* operation, whose periodicity can be controlled to prevent resource exhaustion and etcd performance degradation. Importantly, while compaction does not disrupt normal database operations, during defragmentation, the etcd member rebuilds its state and, hence, it cannot read or write data, leading to service disruption for the MSs that may attempt accessing the NPMs or their state. In our analysis, we account for the service disruption duration, referred to as *defrag downtime*, yielded by each maintenance operation and we assess if, and to what extent, such downtime may impact the ability of the NPM platform to collect performance metrics from the producers and provide them to the consumers.

Redis is a high-performance, in-memory data store with built-in support for persistence, replication, and complex data structures, such as strings, hashes, lists, sets, and JSON objects. Redis provides three main deployment options: standalone, sentinel, and cluster. Standalone Redis represents the centralized architecture, which cannot meet data reliability requirements and does not scale with an increasing demand for computational resources. Redis sentinel, instead, offers a distributed solution with fault tolerance by adding replica nodes. However, it still relies on a single master in charge of read/write operations, limiting scalability and imposing potential bottlenecks. Redis cluster solves the above issues, since it features a fully distributed setup with multiple masters, replicas, and data sharding. Each shard of data is managed by a different master, thus balancing compute load and storage bloat across the cluster, and at least one replica is assigned to each master to provide fail-over capabilities. Since Redis sentinel is primarily a redundancy solution and does not bring scalability benefits, in our analysis we will focus on comparing the standalone and cluster configurations, highlighting the trade-offs in resource usage, availability, and scalability between these Redis architectures for the definition of an NPMs platform.

Importantly, although Redis does not maintain a history of its keyspace, it can still experience memory fragmentation over time due to the memory pages that were allocated by the operative system but cannot be fully utilized by Redis. To address this issue, Redis implements a defragmentation process that periodically reclaims fragmented memory to prevent eventual resource exhaustion. Unlike etcd, however, Redis's defragmentation process is inherently designed to run in the background, hence yielding no service disruption by design. To assess whether and to what extent such a defragmentation process affects resource consumption and the latency of Redis operations, our analysis in Section 6 addresses both operational scenarios, i.e., with defragmentation enabled and disabled.

NPM platform implementation. We now discuss the most relevant data types and APIs provided by etcd and Redis that can be used to realize the NPM platform.

When etcd is selected as platform database, the fundamental data type is the KV pair. The key can be used in a tree-structure format to accurately organize each data sample, e.g., indicating its kind, identification number, and timestamp. The value can be used to store the data of interest either as a plain string or using more complex formats like JSON. By doing so, all data of the NPM platform, i.e., metrics, entries of the MS states, and control actions, e.g., RAN policies and end-devices commands, can be accessed through `get`, `put`, and `del` APIs to, respectively, read, write, and delete them. Importantly, the `watch` API can be used to implement callback routines that are asynchronously triggered whenever a specific key is updated, e.g., the control key for the RAN and the metrics key for the MSs.

Similarly, when Redis is selected as database, the same approach can be used to organize data (i.e., relying on `get`, `set`, and `del` APIs to, respectively, read, write, and delete the KV pairs), since it is essentially a KV store too. However, since Redis does not offer an API equivalent to etcd's `watch`, an alternative approach must be used to implement interrupt-driven routines triggered by KV updates. We address this problem by using *Redis streams* – a data structure optimized for real-time event tracking with high-efficiency insertions and reads via the `xadd` and `xread` APIs. Notice that Redis streams allow the subscription to new elements that are published to the stream, i.e., notifying a client whenever a key is updated. Also, despite being similar to a publish/subscribe approach, Redis streams support multiple clients, i.e., new elements can be delivered to all consumers of the same group, with each consumer processing data at its own pace.

4. The PACE emulation framework

To experimentally compare the centralized and distributed architectures for an NPM platform and assess the performance under varying consistency levels, we designed and developed PACE, a Producer And Consumer Emulator that captures a wide range of interaction patterns and computing load dynamics. We remark that PACE is publicly available in [5]. We introduce PACE components in Section 4.1 and the information flow it implements in Section 4.2.

4.1. Framework components

We consider the same reference scenario as in Section 3.1, where the RAN periodically generates NPMs and edge MSs consume these metrics, using them to produce control policies and update their own internal states accordingly. To synthetically reproduce these entities and their interaction, PACE emulates P producers and C consumers, all implemented as MSs and operating in configurable, arbitrarily paired groups. Each producer can be individually configured in terms of metrics generation period and size, i.e., how frequently and how large the values of the metrics are produced. Upon receiving such metrics, consumers can process them, generate a control message with tunable size, and update their own internal state accordingly. Each consumer's state is implemented with a queue of customizable length and it is uniquely identifiable by the consumer's ID. Also, to create groups of producers and consumers communicating independently from each other and with different settings, a group ID is assigned to both producers and consumers. To seamlessly integrate PACE with etcd and Redis database technologies, we designed it to leverage the etcd and Redis data types and interfaces, as detailed in Section 3.2.

By adjusting the rate of data production and the number of concurrent producer(s)-consumer(s) groups, our emulator is able to encompass different scales of edge networks, from moderate to high-traffic conditions. We leverage on such ability of PACE (see Section 6), to experimentally analyze latency and scalability of the different NPM platform architectures, accounting for the consistency vs. availability trade-off, and under varying traffic load.

4.2. Information flow

We now describe the information flow of PACE and how it encompasses both the centralized and distributed NPM platform implementations we discussed in Section 3.2. We do so in Figs. 4 and 5, showing how to realize such information flow using, respectively, etcd and Redis APIs.

Both diagrams illustrate the flow between a producer MS and a consumer MS, and the NPM platform interfaces they use to interact and to update their internal state. Importantly, each MS is assigned a private ID and a group ID, which allow us to easily extend the presented information flow to scenarios with multiple producers and consumers. In Fig. 4, where etcd is chosen as NPM platform database, both producer and consumer are designed to encapsulate data samples using KV pairs and access them through the corresponding APIs, namely, `put`, `get`, and `del`, to, respectively, write, read, and delete such KV pairs.

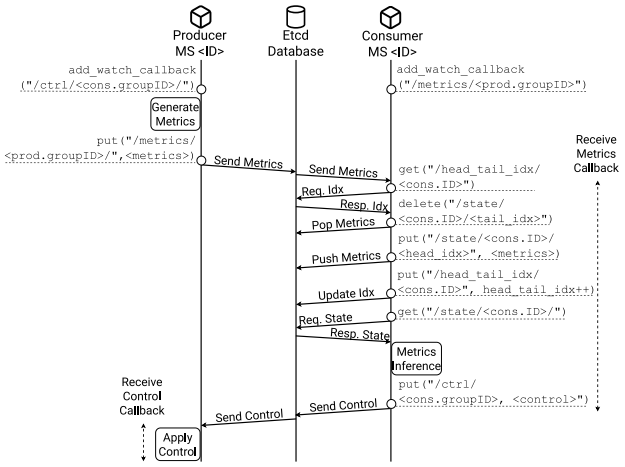


Fig. 4. PACE information flow with etcd APIs.

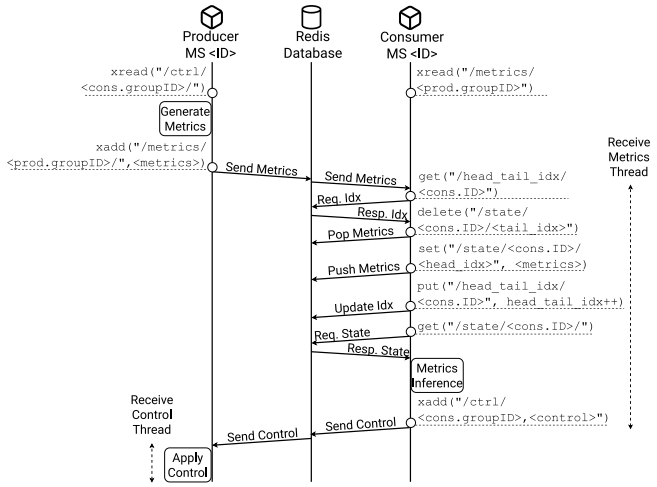


Fig. 5. PACE information flow with Redis APIs.

Keys are organized using a tree structure: `/metrics/` for the NPMs, `/ctrl/` for the control policy, and `/state/` for the state queue. Etcd watch API is then used to notify producers and consumers upon changes of a specific key and execute a callback routine accordingly. To this end, metrics and control keys are organized using the group ID, thus identifying which subset of metrics/control keys each consumer/producer should track. Upon connecting to etcd and enabling the `watch` callback, the producer generates metrics with tunable size and arbitrary frequency and stores them on etcd using the appropriate key. As the value corresponding to the metrics key changes, the consumer receives such new value and starts its callback routine, consisting of the following steps: the consumer (1) retrieves the head and tail indices of its state queue, (2) removes the oldest entry from the tail of the queue, (3) appends the new metrics value at the head of the queue, (4) increments the head and tail indices, (5) retrieves the whole state queue, (6) uses such state queue to execute an inference task, and (7) updates the control key with the result. Eventually, the producer receives the updated control policy via its watch callback and applies it in the corresponding routine.

Similarly, Fig. 5 shows the information flow when Redis is used as NPM platform database, accounting for the specific capabilities and APIs offered by Redis natively. Basic operations like reading, writing,

and deleting KV pairs are handled via `get`, `set`, and `del` APIs. However, unlike etcd, Redis does not offer a direct callback API to track key changes. Therefore, to overcome this issue, we use Redis streams data type with `xread` and `xadd` APIs to, respectively, read and write values to the stream, and with the former acting as a blocking operation that asynchronously waits for new entries. To handle this without blocking other tasks, we also leverage Redis multithreading support to execute `xread` in a separate thread. By doing so, an equivalent flow to the etcd one can be defined: upon connecting to Redis and starting the `xread` thread, the producer generates metrics with customizable size and frequency and updates the corresponding metrics stream. When new metrics are added to the stream, the consumer's thread is notified and it executes the following steps: (1) retrieve the state queue indices, (2) remove the oldest entry, (3) append the new metrics, (4) update the indices, (5) access the entire state queue, (6) perform the inference task, and (7) update the control stream with the result. Finally, the producer receives the updated control policy through the control stream and applies it in its dedicated thread.

5. Testbed development and traffic scenarios

In this section, we first describe the testbed we developed and configured to run our experiments (Section 5.1). Then, we introduce our reference traffic scenarios (Section 5.2), based on which we configure our PACE emulator.

5.1. Testbed and settings

Our testbed is built on top of the CrownLabs [41] bare-metal Kubernetes cluster, hosted at the Politecnico di Torino, Italy. The cluster comprises 4 Dell PowerEdge R740x servers, each featuring an Intel Xeon Gold 5120 CPU and 64 GB of RAM. To gather accurate and comprehensive metrics for our performance evaluation, the testbed integrates Prometheus [42] and Kepler [43]. Prometheus is a widely adopted Kubernetes monitoring system that facilitates effective cluster-wide metrics aggregation. Kepler, on the other hand, is a renown framework that uses advanced power models to estimate real-time energy consumption at the pod level (i.e., at the Kubernetes fundamental unit). Given the importance of accurately estimating a system carbon footprint [44], Kepler accounts not only for the active computations but also for idle power, i.e., the static node power. As thoroughly discussed in [45–47], such idle contribution mainly consists of the power related to hardware components, such as motherboard, fans, network interface cards, and other peripherals, as well as the power consumed by the Kubernetes elements that are necessary for the system to be functional, e.g., the Kubelet and the control plane.

To deploy the NPM platform architectures outlined in Section 3.1, we leverage the off-the-shelf versions of etcd [48] and Redis [49] databases, configured via bitnami helm charts with default settings. For the centralized architecture, we deploy a single etcd member and a single Redis master. Instead, for the distributed architecture, we configure the minimum level of replication that guarantees tolerance to node failures: 3 etcd members and 3 Redis masters, with each Redis master being assigned one replica. In fact, both etcd and Redis require a majority of instances to remain operational; hence, to tolerate the failure of a single edge node, at least 3 instances are necessary. Also, notice that, in general, the number of database instances is not meant to vary in real-time, as it depends only on the cluster architecture design and targets the fault-tolerance requirements. Our testbed setup, featuring a real-world cloud computing architecture and fault tolerance guarantees, allows us to accurately emulate realistic edge computing scenarios and evaluate NPM platform performance under varying architectures and traffic conditions.

5.2. Traffic scenarios

To effectively emulate varying edge scenarios, we run PACE producer(s) and consumer(s) as Kubernetes pods and finely configure them to match different traffic loads. Specifically, we consider four representative classes of NPM traffic, each targeting a typical edge computing scenario. As outlined in Table 1, each class is characterized by (i) the size μ of the generated metrics value, and (ii) the periodicity ν with which these metrics are generated. Class *Pulse* targets scenarios with loose control loops and few NPMs (i.e., small total metrics value size), which is typical of Massive Machine-Type Communications (mMTC) applications. Class *Ping* also features few NPMs but with tight control loops, which is in line with Internet of Things (IoT) telemetry requirements. On the other hand, class *Burst* is characterized by a large metrics value size and loose control loops, which is customary for surveillance and analytics applications. Eventually, class *Wave* targets extreme scenarios where control is frequent and many NPMs are processed at the same time, which models time-critical applications, e.g., self-driving Unmanned Aerial Vehicles (UAVs), requiring low latency control. Also, to account for stateful information, i.e., context-based data each consumer needs to accomplish its tasks, we further configure PACE consumers with a fixed value of state size equal to 1 MB. This value corresponds to a different NPMs retention window depending on the traffic class: for Pulse or Ping traffic, the consumer retains NPMs for approximately 15 min, while, for Burst or Wave traffic, such retention window is reduced to 1 min. In other words, by modeling different traffic patterns and accounting for stateful metrics retention, our setup effectively captures a broad spectrum of real-world scenarios, ensuring a robust evaluation of an edge NPM platform's capabilities and limitations.

Table 1
Representative classes of NPM traffic scenarios.

Class/Characteristics	Pulse	Ping	Burst	Wave
Metrics value size, μ	1 kB	1 kB	100 kB	100 kB
Metrics periodicity, ν	1 s	100 ms	1 s	100 ms
Target scenario	mMTC	IoT	Analytics	UAVs

6. Performance evaluation

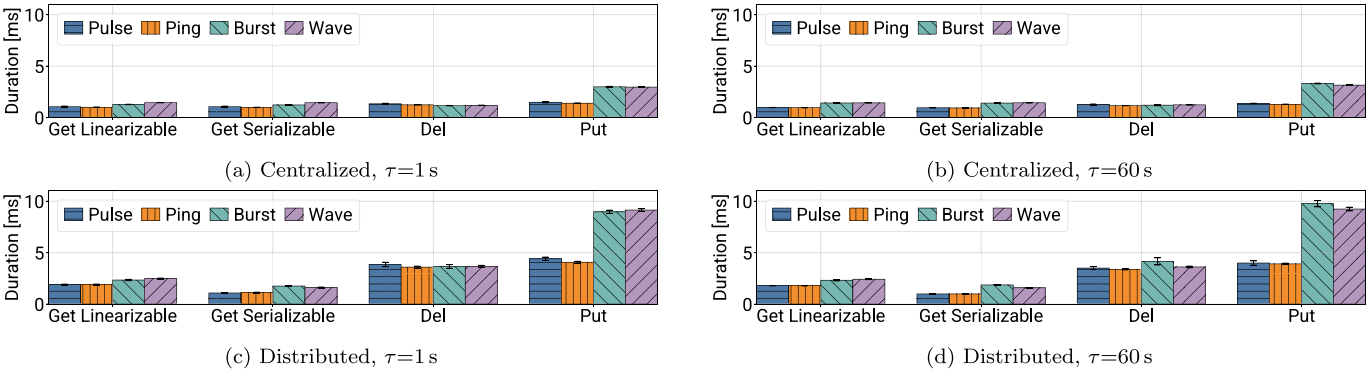
We now present our experimental characterization of NPM platform architectures under various representative traffic classes that we emulated using PACE. Specifically, we first compare the centralized and distributed architectures, for both etcd and Redis implementations, in a single producer–consumer scenario. Then, we compare etcd and Redis in larger-scale scenarios, i.e., emulating multiple producers and multiple consumers across varying traffic classes. Our results cover multiple performance metrics, including database latency, CPU and memory usage, energy consumption, and disk and bandwidth utilization. Such results have been obtained by averaging over at least 200 samples and are reported with 95% confidence interval.

6.1. Centralized vs. distributed architectures

Here we compare the performance between centralized and distributed architectures when using etcd and Redis.

6.1.1. Etcd-based architectures

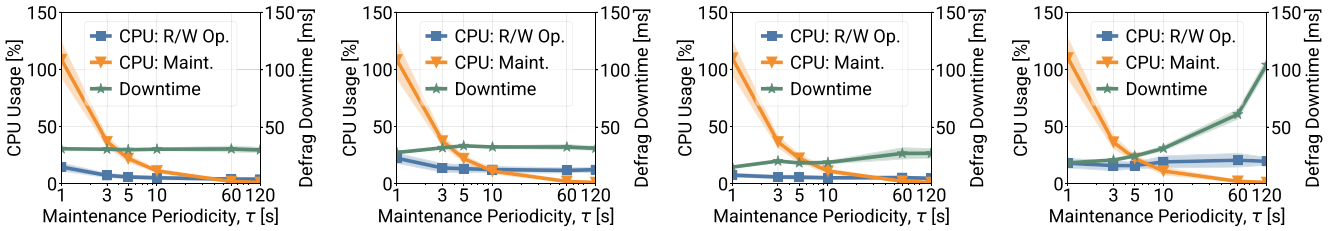
Fig. 6 presents the duration of the fundamental etcd operations, i.e., `get`, `put`, and `del`, for centralized and distributed NPM platforms, under varying traffic classes and maintenance periodicity τ. Importantly, while measuring the `get` duration, we account for the two consistency levels offered by etcd: *linearizable*, which ensures that the returned value is acknowledged by the majority of the etcd members, and *serializable*, which retrieves values without relying on the consensus algorithm. Results show that the duration of the `get` and `put` operations depends on the traffic class and, in particular, exhibits a positive correlation with the metrics value size μ while it is independent of the generation periodicity ν. For instance, the Wave traffic class



(a) Centralized, τ=1 s (b) Centralized, τ=60 s

(c) Distributed, τ=1 s (d) Distributed, τ=60 s

Fig. 6. Duration of the etcd APIs under the centralized (a–b) and distributed (c–d) NPM platform architectures and for varying values of etcd maintenance periodicity τ.



(a) Pulse (b) Ping (c) Burst (d) Wave

Fig. 7. Etcd-based, centralized platform's CPU usage for read/write and maintenance operations, and defrag downtime vs. maintenance periodicity τ for different traffic classes.

(μ = 100 kB) shows significantly higher latency than Pulse (μ = 1 kB), while classes with different periodicity, such as Pulse (ν = 1 s) and Ping (ν = 0.1 s), exhibit a similar behavior. In contrast, the `del` operation is not affected by either μ or ν.

Next, comparing the obtained latency for different values of τ, it is evident that maintenance operations have negligible impact. Instead, the comparison between the two architectures underscores that the distributed setup consistently incurs higher latency. This effect is most pronounced for the `put` operation, due to the communication overhead introduced by the consensus model. For the same reason, the linearizable `get` operation demonstrates longer duration than the serializable one, as the former involves multiple round-trip times to achieve consensus while the latter does not.

Focusing on the impact of etcd maintenance operations, Fig. 7 analyzes the centralized platform's CPU usage and defrag downtime as functions of τ and varying traffic classes. We remark that these results align with those of the distributed platform on a per-member basis. We identify two fundamental CPU usage contributions: one for the normal read/write database operations and the other for maintenance operations. The defrag downtime measures the maximum service disruption duration experienced by a client when attempting to contact a specific etcd member that is temporarily blocked by an ongoing maintenance operation (see Section 3.2). Notably, regardless of the traffic class, maintenance operations dominate the computing resources consumption, with values that are negatively correlated with τ, reaching up to 100% when τ = 1 s. In contrast, the CPU usage of R/W operations is practically independent of τ, while it is affected by the traffic class and, specifically, by the metrics period ν. In fact, while no significant difference can be observed between Pulse and Burst or between Ping and Wave, when we decrease ν from 1 s to 100 ms, a higher CPU usage is observed.

Looking at the defrag downtime, it can be seen that, for the Pulse, Ping, and Burst classes, τ has negligible impact and, instead, for the Wave class, the downtime increases strongly with τ. This is due to the fact that Wave is characterized by large and frequent NPMs, which yields high database fragmentation and, consequently, a higher service disruption duration. We conclude that, depending on the traffic class of interest, there is a trade-off between total CPU consumption and defrag downtime: when τ is not relevant to the downtime, larger values can be selected, thus minimizing the consumption of compute resources. Instead, when τ impacts downtime, an optimal value can be determined to minimize both resource usage and defrag downtime.

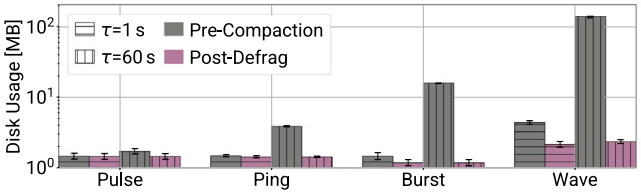


Fig. 8. Etcd-based, centralized platform's disk usage before and after maintenance operations, for different traffic classes and varying values of maintenance periodicity τ. These results align with those of the distributed platform on a per-member basis.

To examine the effect of periodic maintenance operations on disk usage, Fig. 8 compares the centralized platform's pre-compaction and post-defragmentation disk usage across traffic classes and for varying values of τ. We remark that these results align with those of the distributed platform on a per-member basis. Pre-compaction values represent the maximum amount of disk space allocated by etcd to store the complete revision history since the last maintenance operation, while post-defragmentation values reflect the minimum amount of space needed to store only the latest data revision, i.e., upon clearing such history of changes and retaining the most meaningful data. Thus, as expected, pre-compaction disk usage always exceeds the post-defragmentation ones, and the gap between the two widens with the traffic demand and with the maintenance periodicity. For instance, while with Pulse, regardless of τ, no significant gap between pre-compaction and post-defrag can be observed, in the case of Wave such gap is not negligible and becomes pronounced at τ = 60 s, reaching values up to 10^2. The reason for this behavior is threefold: (i) the larger the μ, the larger each entry in the revision history; (ii) the smaller the ν, the greater the number of entries in the revision history; (iii) the larger

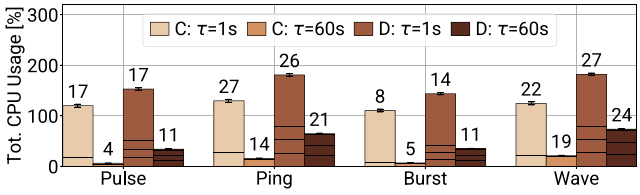


Fig. 9. Etcd-based, centralized and distributed platforms' stacked CPU usage of each member, for different traffic classes and varying values of maintenance periodicity τ. Text indicates the average usage of a member during normal operations.

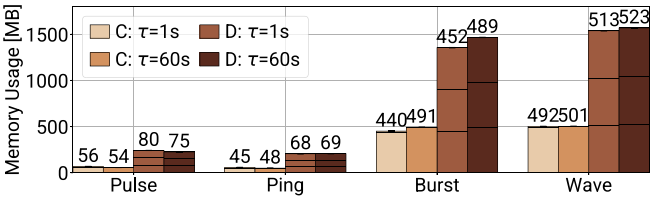


Fig. 11. Etcd-based, centralized and distributed platforms' stacked memory usage of each member, for different traffic classes and varying values of maintenance periodicity τ.

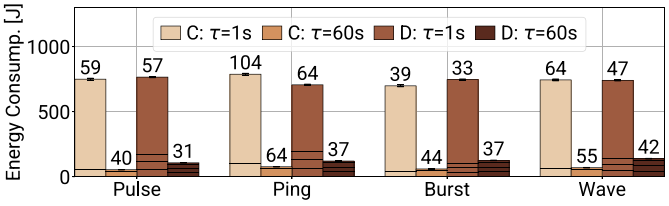


Fig. 10. Etcd-based, centralized and distributed platforms' stacked energy consumption of each member, for different traffic classes, varying values of maintenance periodicity τ, and considering a 30-s time window. Text indicates the average consumption of a member during normal operations.

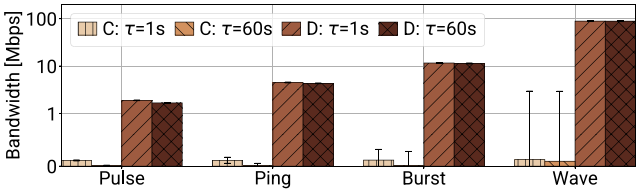


Fig. 12. Etcd-based, centralized and distributed platforms' bandwidth consumption, for different traffic classes and varying values of maintenance periodicity τ.

the τ, the larger the amount of changes that are accumulated. These results, along with those in Fig. 7, further emphasize the importance of carefully selecting the value of τ, which not only impacts resource usage and downtime but can also yield resource exhaustion if not properly controlled.

To compare the consumption of compute resources between the centralized and distributed architectures, we analyze the total CPU usage (i.e., including both R/W and maintenance components) in Fig. 9, accounting for different traffic classes and varying values of τ. Each bar represents the stacked contribution of individual etcd members and, above each bar, we annotate the value of CPU usage for R/W operations normalized with respect to the number of etcd members, i.e., $r = 1$ ($r = 3$) in the centralized (distributed) case. In fact, results show that while varying the traffic class has negligible impact on the total CPU usage, it affects the R/W CPU usage, with Ping and Wave being the most compute demanding classes due to their small value of ν. As expected, comparing the two architectures, the distributed one always requires a higher amount of compute resources, regardless of τ and the traffic class, which is due to the increase in the number of etcd members, all requiring periodic maintenance operations and participating jointly in each read/write operation according to the consensus model.

Fig. 10 illustrates the energy consumption of the centralized and distributed architectures, measured over a 30-s window and accounting for varying values of τ and different traffic classes. As above, each bar is a stack of the etcd members' contributions, and, to highlight the impact of maintenance operations, we report the value of energy consumed by each member for its R/W operations. Consistently with the results for CPU usage, the total energy consumption is practically independent of the traffic class and is mostly dominated by maintenance operations. In fact, when $\tau = 1$ s, centralized and distributed architectures demonstrate comparable results, as maintenance operations dominate the overall usage of resources. In contrast, at $\tau = 60$ s, the distributed architecture exhibits slightly higher resource consumption, due to the larger number of etcd members. Also, looking at the consumption per instance, there is minimal variation across classes and regardless of τ, with the only exception of Ping in the centralized case with $\tau = 1$ s. The latter indeed shows a higher value due to saturation caused by the large number of NPMs being retained and frequently updated.

Memory usage, depicted in Fig. 11 as a function of the traffic class, for varying values of τ, depends primarily on the value of μ. In fact, while varying ν has no impact, increasing μ from 1 kB to 100 kB (e.g., from Ping to Wave) results in a significant growth of memory usage, reaching up to 2 GB. Further, memory usage is higher under the distributed architecture, due to the increased number of etcd members, and it is almost independent of τ, since maintenance operations have no impact on the memory, which is used by etcd as a cache to quickly look up KV pairs. Also, comparing the total memory usage with the individual member contribution, we conclude that such usage scales consistently with the number of etcd members r.

Finally, we compare bandwidth consumption between centralized and distributed architectures in Fig. 12, for different traffic classes and varying values of τ. Specifically, we measure the bandwidth consumption that is exclusively related to database operations, i.e., we exclude the predictable traffic sent/received by PACE. By doing so, we highlight how much bandwidth must be reserved for the communication between the edge nodes for the consensus model to be functional. As expected, since consensus is disabled on the centralized architecture, the measured bandwidth is approximately equal to zero. Instead, looking at the results for the distributed architecture, we notice that the bandwidth consumption is non-negligible and can reach up to a 100 Mbps in the case of the most demanding traffic class. This clearly imposes scalability challenges as multiple instances of demanding traffic classes may contribute to saturation of inter-node bandwidth resources.

6.1.2. Redis-based architectures

We now continue with our comparison between centralized and distributed NPM architectures by considering their Redis implementation. Also, to analyze the impact of Redis defragmentation process on database latency and resource consumption, we report results with enabled (δ = ON) and disabled (δ = OFF) defragmentation.

Fig. 13 depicts the duration of the fundamental Redis operations, i.e., `get`, `set`, and `del`, for different NPM traffic classes and varying defragmentation state δ. Results show that, while the duration of the `get` and `set` operations depends on the traffic class and is positively correlated with μ, although independent of ν, the `del` operation is practically unaffected by either μ or ν. When comparing the two architectures, the `set` operation nearly doubles in duration when Redis is distributed, primarily due to sharding and data replication. Further, we notice that, regardless of the architecture, none of the operations are affected by the defragmentation process, as varying δ yields negligible effects on the operations duration.

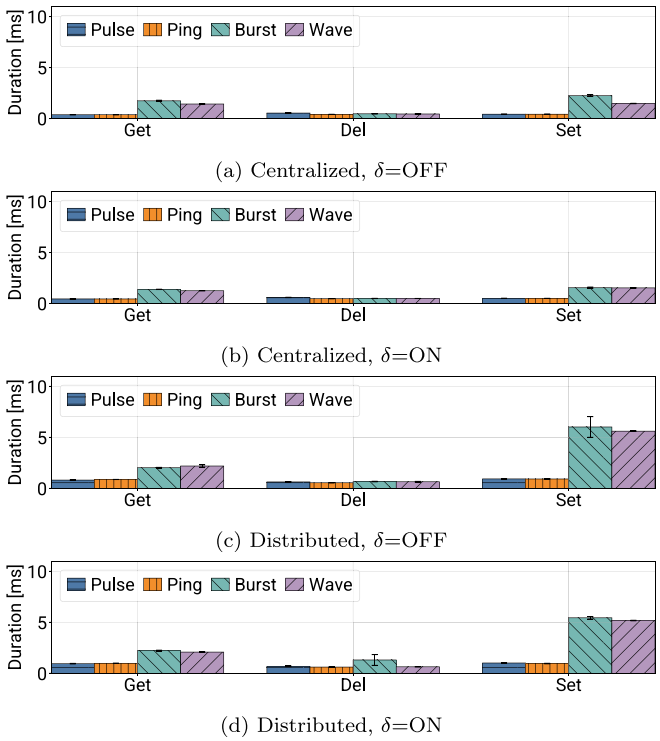


Fig. 13. Duration of the Redis APIs under the centralized (a-b) and distributed (c-d) NPM platform architectures for different traffic classes and varying defragmentation state δ.

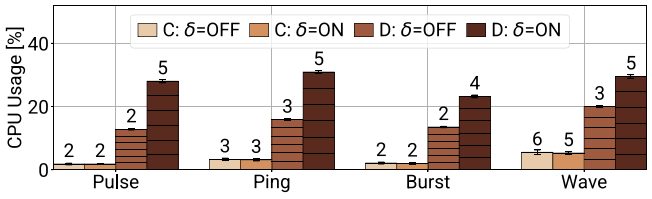


Fig. 14. Redis-based, centralized and distributed platforms' stacked CPU usage of each instance, for different traffic classes and varying defragmentation state δ. Text indicates the average usage of an instance during normal operations.

Looking at the usage of compute resources, Fig. 14 compares the centralized and distributed platforms for different traffic classes and varying defragmentation state δ. Each bar represents the stacked contributions of individual Redis instances. Additionally, the text above each bar indicates the CPU usage normalized by the number of Redis instances, i.e., $r = 1$ ($r = 6$) in the centralized (distributed) case. Results show that the impact of the traffic class is minimal across all configurations, although Ping and Wave exhibit slightly higher resource usage due to their decreased metric generation periodicity ν. As expected, the distributed architecture yields higher resource consumption owing to the increased number of Redis instances. However, the CPU usage per instance remains practically consistent across the two architectures. Notably, δ has no significant effect on the centralized architecture, while in the distributed one it nearly doubles resource consumption due to the defragmentation of multiple Redis instances.

Fig. 15 shows the energy consumption of the centralized and distributed architectures, measured over a 30-s time window, for different traffic classes and defragmentation state δ. Again, each bar represents the stacked contributions of individual Redis instances, and the text above each bar indicates the consumption of each Redis instance. We notice that results are consistent with those for CPU usage: the energy consumption is largely independent of the traffic class and higher under

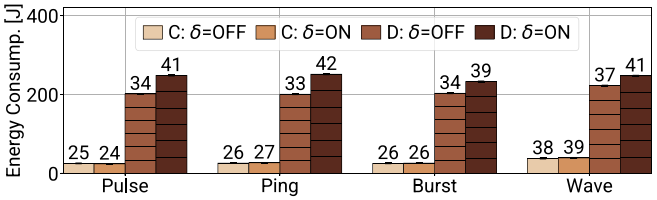


Fig. 15. Redis-based, centralized and distributed platforms' stacked energy consumption of each instance, for different traffic classes, varying defragmentation state δ, and considering a 30-s time window. Text indicates the average consumption of a single instance.

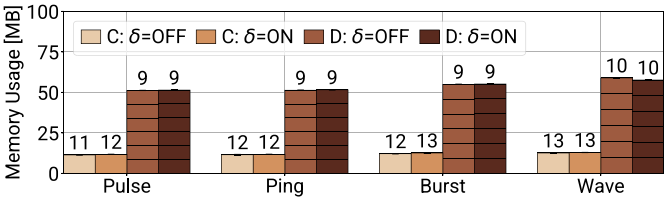


Fig. 16. Redis-based, centralized and distributed platforms' stacked memory usage of each instance, for different traffic classes and varying defragmentation state δ. Text indicates the average usage of a single instance.

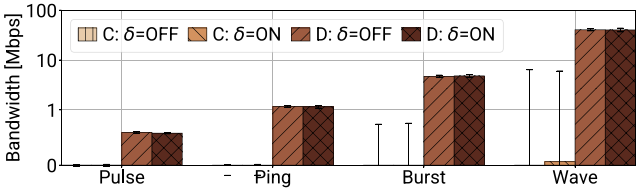


Fig. 17. Redis-based, centralized and distributed platforms' bandwidth usage, for varying traffic classes, and varying defragmentation state δ.

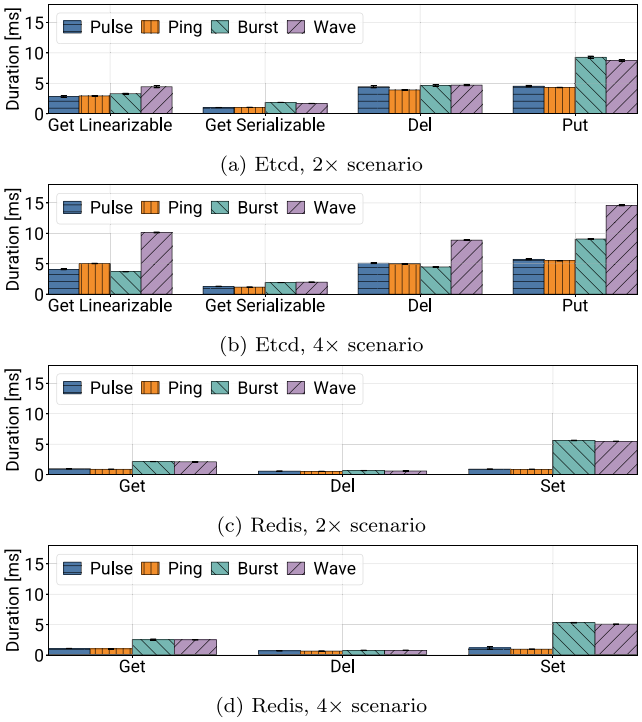
the distributed architecture. Although δ has minimal impact on the centralized setup, it leads to a slight increase in energy consumption in the distributed case, due to the larger number of etcd instances. Also, regardless of the traffic class and δ, the value of energy consumed per instance is practically constant, emphasizing how the total energy consumption scales with the number of instances r.

Stacked memory usage is analyzed in Fig. 16, as a function of the traffic class and for varying defragmentation state δ. Notably, results indicate that the memory usage is practically independent of both the traffic class and δ, which is mainly due to the fact that Redis does not retain old revisions of each KV pair. Instead, comparing the total memory usage with the one related to the individual Redis instance, we observe that such usage scales consistently with the number of Redis instances r.

Finally, Fig. 17 presents the bandwidth consumption for centralized and distributed architectures under varying traffic classes and defragmentation state δ. Specifically, we measure the bandwidth consumption related to internal database operations, i.e., excluding the predictable traffic sent/received by PACE. By doing so, we effectively assess the impact of Redis replication on bandwidth resources. As expected, since the centralized architecture features no replication, no bandwidth is consumed. Conversely, the distributed architecture exhibits increasing bandwidth consumption with the traffic class, reaching up to 40 Mbps in the case of the most demanding class. Also, by varying δ, we observe that defragmentation has no impact on bandwidth consumption.

6.1.3. Major lessons learned

From the above comparison, the following main findings have emerged:



(a) Etcd, 2× scenario

(b) Etcd, 4× scenario

(c) Redis, 2× scenario

(d) Redis, 4× scenario

Fig. 18. Distributed, etcd- and Redis-based platforms' APIs duration in our 2× and 4× scenarios, for varying traffic classes.

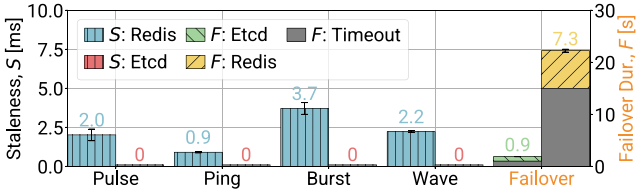


Fig. 19. Staleness of NPMs for varying traffic classes and failover duration for both etcd- and Redis- based distributed platforms in the event of a network partition. The default configurable failover timeout is equal to 1 s and 15 s for etcd and redis, respectively.

• Regardless of the specific implementation, the centralized architecture exhibits lower database operation latency and reduced resource consumption than the distributed one, but at the cost of no fault tolerance and limited scalability;

• For both centralized and distributed architectures, etcd maintenance operations (i) have limited impact on latency, memory, and bandwidth, but (ii) strongly dominate CPU, disk, and energy consumption, and (iii) introduce periodic service disruptions whose duration depends on the traffic class;

• Conversely, Redis defragmentation process yields no service disruption by design and has limited impact on database operation latency and resource consumption;

• By reducing consistency guarantees, the Redis-based distributed architecture achieves up to 50% lower database operation latency and reduced per-instance resource consumption compared to the etcd-based one.

6.2. Scalability of distributed architectures

We now extend our performance evaluation by analyzing the scalability of distributed NPM platforms based on etcd and Redis under increased traffic loads in larger-scale scenarios involving multiple producers and consumers. To do so, we use PACE to emulate two increasingly complex scenarios, namely, 2× with 2 producers, and 4× with 4 producers. In both scenarios, each producer serves 3 consumers, resulting in a total of 12 MSs (16 MSs) in the 2× (4×) scenario, all simultaneously reading and writing NPMs through the platform. Then, we quantitatively illustrate the trade-off between the strong consistency of etcd and the high availability of Redis as well as their robustness against edge node failures. Again, in our experiments, we consider the representative traffic classes outlined in Table 1. Since we already thoroughly covered the impact of maintenance operations, this analysis focuses on comparing etcd and Redis in scenarios where maintenance has minimal influence on resource consumption, i.e., $\tau = 60\,\text{s}$ for etcd and $\delta =$ OFF for Redis.

6.2.1. Operation latency

To assess the scalability of distributed NPM platforms under increased traffic loads, Fig. 18 presents the duration of the fundamental etcd and Redis operations, for varying NPM traffic classes and under the 2× and 4× scenarios.

Focusing on the etcd-based platform (Figs. 18(a)–18(b)), results indicate that the duration of the `get` and `put` operations is positively correlated with the metrics value size μ and that, in the 2× scenario, the impact of the generation periodicity ν is minimal. However, in the 4× scenario, i.e., under higher traffic conditions, decreasing ν significantly affects the operation durations. For instance, for the most demanding traffic class (Wave), operation durations can double compared to the least demanding class (Pulse). In contrast, the `del` operation remains largely unaffected by μ or ν, except for the Wave class in the 4× scenario, where extreme traffic conditions push etcd to saturation. Furthermore, compared to the single producer/consumer scenario in Fig. 6, etcd exhibits a noticeable increase in most operation durations, highlighting that the impact of strong consistency guarantees on availability becomes more pronounced as the traffic load increases. In fact, when consistency guarantees are reduced upon reads through the serializable `get` operation, etcd demonstrates consistent availability levels, regardless of the traffic scenario.

Conversely, in the Redis-based platform (Figs. 18(c)–18(d)), the duration of `get` and `put` operations remains positively correlated with μ but it is independent of ν, regardless of the scenario. Similarly, the `del` operation is unaffected by both μ and ν. Notably, comparing the 2× and 4× scenarios, all operations exhibit negligible variations in duration despite the increased traffic load, remaining up to three times faster than their etcd counterparts. When compared to the single producer/consumer scenario in Fig. 13, Redis maintains its availability guarantees with negligible impact on its operations performance.

Overall, these results demonstrate that both platforms scale well with traffic load, and the operation durations remain compatible even with the tightest control loops (see Section 5.2). Furthermore, Redis ensures consistent availability across varying traffic loads, whereas etcd's availability is progressively penalized by higher traffic intensity due to its enforcement of strong consistency guarantees.

6.2.2. Staleness and failover

To quantitatively illustrate the trade-off between the strong consistency of etcd and the high availability of Redis (see Section 3.2), we analyze two key metrics: *staleness* and *failover duration*. Staleness measures the time during which NPMs are written but not yet replicated across the distributed database, raising concerns about (i) potential data loss during node failures or network partitions, and (ii) clients accessing outdated NPMs during normal operations. Failover duration, on the other hand, quantifies the time required for a Redis replica to assume the role of a new master or for an etcd member to be elected as the new leader, reflecting the service disruption that is experienced by clients in the event of a network partition or edge node failure. Fig. 19 presents staleness and failover duration for distributed NPM platforms implemented with etcd and Redis. By design, etcd ensures

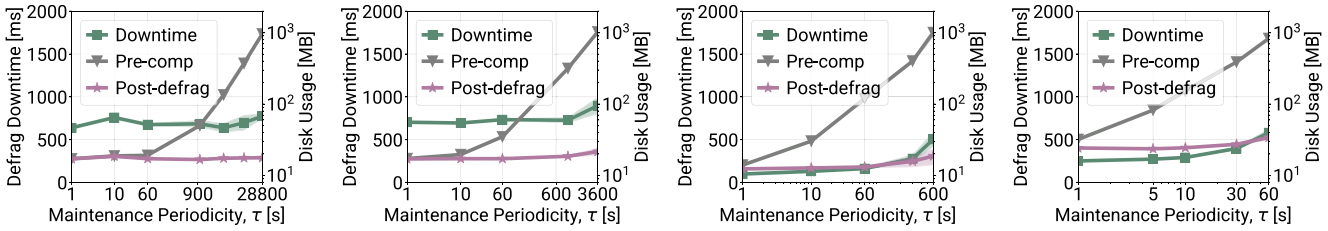


Fig. 20. Defragmentation downtime and disk usage of etcd-based, distributed NPM platforms as functions of the maintenance periodicity τ and for varying traffic classes.

zero staleness, independently from the traffic class, due to the strong consistency provided by its consensus model (see Section 3). Importantly, this guarantees that when the NPM platform is implemented using etcd, clients always access up-to-date NPMs, even during critical network conditions. In contrast, Redis prioritizes availability over consistency through asynchronous replication, resulting in inevitable staleness, which we observe to be in the order of a few milliseconds. Regarding failover, two factors contribute to the observed durations: (i) a configurable timeout before elections, which defaults to 1 s for etcd and 15 s for Redis, and (ii) the time that is practically needed for a new etcd leader or Redis master to take over. While etcd achieves this in less than a second, Redis can take up to 10 s of seconds. We conclude that it is of paramount importance to consider staleness and failover duration in the design of NPM platforms as they may significantly impact service continuity, especially for reliability- and time-critical applications.

6.2.3. Maintenance KPIs

As discussed in Section 3.2 and demonstrated in Section 6.1, while Redis performance and resource usage are not impacted by defragmentation, etcd's strong consistency entails increased disk usage. This is due to revision history retention and the periodic downtime caused by maintenance operations. To observe how these metrics scale in our 4× scenario, Fig. 20 shows defrag downtime and disk usage as functions of the maintenance periodicity τ and for different traffic classes. Specifically, we report disk usage both before and after maintenance, namely, pre-compaction and post-defragmentation. Results indicate that defrag downtime increases with τ and varies with the traffic class: higher fragmentation in Pulse and Ping classes (due to smaller μ values) results in longer downtime compared to Burst and Wave. Then, disk usage depends on both τ and the traffic class: as τ grows, the gap between pre-compaction and post-defragmentation increases up to 1 GB, with the steepness of such increase depending on the traffic class. Also, comparing these results to those in Figs. 7 and 8, we observe that both defrag downtime and disk usage scale proportionally with the total number of producers and consumers, as values are approximately 15 times larger than the single producer–consumer scenario. This has two critical implications: (i) shorter maintenance intervals (τ) amplify the relative impact on service continuity, e.g., at $\tau = 1$ s, a defrag downtime of 500 ms may disrupt the service up to 50% of the time; (ii) etcd enforces a default disk usage limit of 2 GB to prevent performance degradation, halting database operations until maintenance is performed. In fact, while in the case of Pulse such a limit is reached after a very long time (namely, 8 h), under Wave traffic stability is compromised after just 60 s. Therefore, we conclude that it is of pivotal importance to optimally select the value of τ in a way that jointly minimizes service disruption, ensures stable performance, and prevents resource exhaustion.

6.2.4. Resource usage

Fig. 21 compares the CPU usage of the etcd and Redis-based distributed NPM platforms under various traffic classes, and under both the 2× and 4× scenarios. Above each bar, we annotate the average CPU usage per etcd member or Redis instance. Results demonstrate that

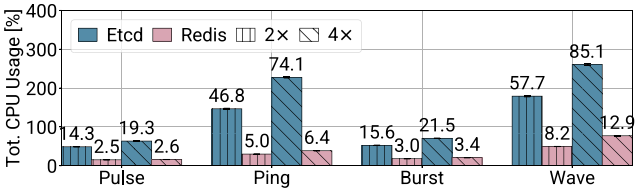


Fig. 21. Distributed, etcd- and Redis-based platforms' CPU usage in our 2× and 4× scenarios, for varying traffic classes. Text indicates the average usage of each etcd member/Redis instance.

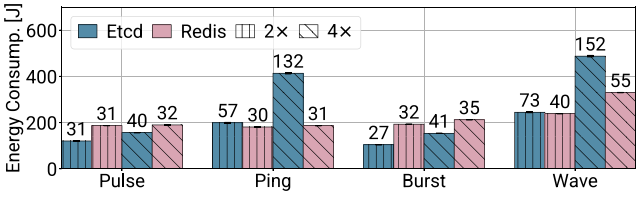


Fig. 22. Distributed, etcd- and Redis-based platforms' energy consumption in our 2× and 4× scenarios, for varying traffic classes. Text indicates the average consumption of each etcd member/Redis instance.

CPU usage significantly depends on the traffic class and, specifically, on ν: classes with frequent metrics generation (Ping and Wave, with $\nu = 100$ ms) yield a significant CPU demand — up to 4 times higher than less demanding classes (Pulse and Burst, with $\nu = 1$ s). Also, regardless of the traffic class, we observe that etcd exhibits consistently higher CPU consumption than Redis, with values from 2 to 4 times higher, reflecting the computational cost of maintaining strong consistency through its consensus model. Comparing these results to the single producer/consumer scenario in Figs. 9 and 14, etcd and Redis maintain comparable values of CPU usage, indicating that both platforms scale effectively in terms of compute resource consumption.

Fig. 22 illustrates the energy consumption of etcd and Redis-based distributed NPM platforms, as a function of the traffic class and under both the 2× and 4× scenarios. Again, text above bars indicate the energy consumed per etcd member/Redis instance. Consistently with the trends observed for CPU usage, energy consumption depends on the traffic class and is mostly affected by ν, i.e., it significantly grows as ν decreases. Such an increase reaches up to 4 times in the case of etcd, while Redis, instead, demonstrates a more uniform energy profile across traffic classes. Notably, while Redis achieves significantly lower CPU usage, this difference does not always translate into reduced per-instance energy consumption, which remains comparable in some cases, such as for the Pulse and Burst classes. In fact, as discussed in Section 5.1, Kepler accounts for both active (computation-related) power consumption and idle power consumption, which includes the power related to additional hardware and software components required for the Kubernetes system to operate. When compared to the single producer/consumer scenario in Figs. 10 and 15, etcd shows

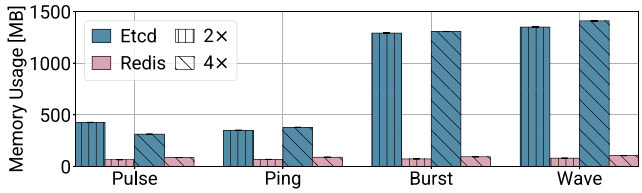


Fig. 23. Distributed, etcd- and Redis-based platforms memory usage in our 2× and 4× scenarios, for varying traffic classes.

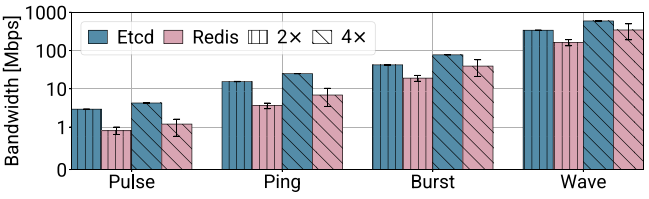


Fig. 24. Distributed, etcd- and Redis-based platforms bandwidth usage in our 2× and 4× scenarios, for varying traffic classes.

a significant increase in energy consumption for the Ping and Wave classes, while Redis attains consistent values, indicating that Redis may offer better scalability than etcd in energy-critical deployments.

Fig. 23 presents the memory usage of distributed etcd and Redis-based NPM platforms, for different traffic classes, and for both the 2× and 4× scenarios. When etcd is used, memory usage depends on the traffic class, and, specifically, on μ, with Burst and Wave (μ = 100 kB) classes yielding up to 3 times higher consumption than Pulse and Ping (μ = 1 kB). On the other hand, traffic classes have no such an impact with Redis, since Redis does not retain a revisions history. Interestingly, these results are similar to those obtained in the single producer/consumer scenario (Figs. 11 and 16), confirming that both platforms efficiently manage memory consumption, even when the number of producers and consumers increase.

Fig. 24 compares the bandwidth consumption of the etcd and Redis-based distributed NPM platforms, for different traffic classes and both the 2× and 4× scenarios. We report the amount of bandwidth that is consumed exclusively for database operations, i.e., excluding the predictable traffic sent/received by PACE. By doing so, we isolate the impact that the etcd consensus model and Redis asynchronous replication entail on the bandwidth resources allocated on the link between edge nodes. Results demonstrate that etcd consistently consumes more bandwidth than Redis, regardless of the traffic class. This is because the former features a fully distributed architecture and suffers from the overhead of the consensus model, while the latter leverages data sharding and asynchronous replication. Importantly, these results are approximately 15 times higher than in the single producer/consumer scenario (Figs. 12 and 17), which is consistent with the increase in the number of producers and consumers.

6.2.5. Major lessons learned

From the above scalability analysis of distributed architectures, the following can be inferred:

- Both etcd- and Redis-based platforms exhibit good operations performance under increased traffic loads, with Redis maintaining consistent availability and etcd experiencing growing, yet acceptable, performance penalties due to its strong consistency guarantees;
- Compared to Redis, etcd ensures zero staleness and shorter failover duration, making etcd better suited for reliability- and time-critical applications, with greater resilience to node failures and network partitions;
- Etcd service disruption duration and disk usage scale well with the number of producers and consumers and are affected by both the maintenance periodicity and the traffic scenario. Proper tuning of the maintenance periodicity is thus essential to prevent resource exhaustion and performance degradation;
- Both etcd- and Redis-based platforms demonstrate a good level of scalability of resource usage with the number of producers and consumers, although Redis does so more efficiently, with consistently lower per-instance values across all traffic classes.

7. Conclusions

Microservice-based architectures have emerged as the most effective application design paradigm in edge computing, delivering high scalability and sustained performance for real-time data processing and AI-driven optimization tasks. Nevertheless, several technical challenges related to microservice coordination, data sharing, and consistency requirements call for innovative data management solutions that go beyond traditional loosely coupled databases. To fill such gaps, we envisioned a central, holistic platform that jointly coordinates cluster-wide collection and usage of network performance metrics (NPM) while decoupling microservices from their internal state. Central to our work is PACE, a finely tunable, microservice-based, emulation framework of NPMs producers and consumers that we developed to fully capture various interaction patterns and realistic traffic dynamics at scale. Through PACE we thoroughly characterized different NPM platform architectures and implementations based on off-the-shelf database solutions across a spectrum of scenarios, ranging from loosely controlled loops with minimal NPMs processing, to latency-critical and compute-demanding use cases.

Our results demonstrate that PACE effectively sheds light on the fundamental trade-offs in performance, scalability, resource usage, and energy efficiency inherent in these architectures and implementations. We noted that some of these excel in ensuring scalability, availability, and efficient resource utilization, while others are indispensable when resilience and strong data consistency are of utmost importance. Further, we showed that careful tuning of these implementations is essential to adapt to specific traffic scenarios without compromising system performance or stability. We thus conclude that PACE proves to be an effective tool for identifying the most suitable solutions based on the edge scenario and the required levels of data consistency and reliability, supporting the design of robust and efficient MSs coordination and data sharing at the network edge.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: The declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work was funded through the 6G-INTENSE project (https://6g-intense.eu/), which has received funding from the Smart Networks and Services Joint Undertaking (SNS JU) under the European Union's Horizon Europe research and innovation programme under Grant Agreement No. 101139266, and through the CSI-Future project under the European Union - Next Generation EU under the Italian National Recovery and Resilience Plan (NRRP) PRIN 2022 program (D.D.1409 del 14/09/2022 MUR). Views and opinions expressed are however those of the authors only and do not necessarily reflect those of the MUR and European Union.

Data availability

No data was used for the research described in the article.

References

[1] S. Wang, Y. Guo, N. Zhang, P. Yang, A. Zhou, X. Shen, Delay-aware microservice coordination in mobile edge computing: A reinforcement learning approach, IEEE Trans. Mob. Comput. 20 (3) (2021) 939–951, http://dx.doi.org/10.1109/TMC.2019.2957804.

[2] R. Laigner, Y. Zhou, M.A.V. Salles, Y. Liu, M. Kalinowski, Data management in microservices: state of the practice, challenges, and research directions, Proc. VLDB Endow. 14 (13) (2021) 3348–3361, http://dx.doi.org/10.14778/3484224.3484232.

[3] N. Mateus-Coelho, M. Cruz-Cunha, L.G. Ferreira, Security in microservices architectures, Procedia Comput. Sci. 181 (2021) 1225–1236, http://dx.doi.org/10.1016/j.procs.2021.01.320, CENTERIS 2020 - International Conference on ENTERprise Information Systems / ProjMAN 2020 - International Conference on Project MANagement / HCist 2020 - International Conference on Health and Social Care Information Systems and Technologies 2020, CENTERIS/ProjMAN/HCist 2020. URL https://www.sciencedirect.com/science/article/pii/S1877050921003719.

[4] A. Rezaei Nasab, M. Shahin, S.A. Hoseyni Raviz, P. Liang, A. Mashmool, V. Lenarduzzi, An empirical study of security practices for microservices systems, J. Syst. Softw. 198 (2023) 111563, http://dx.doi.org/10.1016/j.jss.2022.111563, URL https://www.sciencedirect.com/science/article/pii/S0164121222002394.

[5] A. Calagna, S. Ravera, C.F. Chiasserini, Pace, 2024-2025, https://github.com/antoniocalagna98/PACE.

[6] Etcd team, A distributed, reliable key-value store for the most critical data of a distributed system, 2013-2024, https://etcd.io and https://github.com/etcd-io/etcd.

[7] Redis team, An in-memory database that persists on disk, 2020-2024, https://redis.io and https://github.com/redis/redis.

[8] M.D. Hossain, T. Sultana, S. Akhter, M.I. Hossain, N.T. Thu, L.N. Huynh, G.-W. Lee, E.-N. Huh, The role of microservice approach in edge computing: Opportunities, challenges, and research directions, ICT Express 9 (6) (2023) 1162–1182, http://dx.doi.org/10.1016/j.icte.2023.06.006, URL https://www.sciencedirect.com/science/article/pii/S2405959523000760.

[9] F. Al-Doghman, N. Moustafa, I. Khalil, N. Sohrabi, Z. Tari, A.Y. Zomaya, AI-enabled secure microservices in edge computing: Opportunities and challenges, IEEE Trans. Serv. Comput. 16 (2) (2023) 1485–1504, http://dx.doi.org/10.1109/TSC.2022.3155447.

[10] M. Yao, M. Sohul, V. Marojevic, J.H. Reed, Artificial intelligence defined 5G radio access networks, IEEE Commun. Mag. 57 (3) (2019) 14–20, http://dx.doi.org/10.1109/MCOM.2019.1800629.

[11] P. McEnroe, S. Wang, M. Liyanage, A survey on the convergence of edge computing and AI for UAVs: Opportunities and challenges, IEEE Internet Things J. 9 (17) (2022) 15435–15459, http://dx.doi.org/10.1109/JIOT.2022.3176400.

[12] Z. Liu, C. Zhan, Y. Cui, C. Wu, H. Hu, Robust edge computing in UAV systems via scalable computing and cooperative computing, IEEE Wirel. Commun. 28 (5) (2021) 36–42, http://dx.doi.org/10.1109/MWC.121.2100041.

[13] P. Amanatidis, D. Karampatzakis, G. Michailidis, T. Lagkas, G. Iosifidis, Adaptive reverse task offloading in edge computing for AI processes, Comput. Netw. 255 (2024) 110844, http://dx.doi.org/10.1016/j.comnet.2024.110844, URL https://www.sciencedirect.com/science/article/pii/S1389128624006765.

[14] B. Li, W. Liu, W. Xie, X. Li, Energy-efficient task offloading and trajectory planning in UAV-enabled mobile edge computing networks, Comput. Netw. 234 (2023) 109940, http://dx.doi.org/10.1016/j.comnet.2023.109940, URL https://www.sciencedirect.com/science/article/pii/S1389128623003857.

[15] Q. Zhang, L. Gui, F. Hou, J. Chen, S. Zhu, F. Tian, Dynamic task offloading and resource allocation for mobile-edge computing in dense cloud RAN, IEEE Internet Things J. 7 (4) (2020) 3282–3299, http://dx.doi.org/10.1109/JIOT.2020.2967502.

[16] Q. Yang, S.-C. Chu, C.-C. Hu, L. Kong, J.-S. Pan, A task offloading method based on user satisfaction in C-RAN with mobile edge computing, IEEE Trans. Mob. Comput. 23 (4) (2024) 3452–3465, http://dx.doi.org/10.1109/TMC.2023.3275580.

[17] S. Dong, J. Tang, K. Abbas, R. Hou, J. Kamruzzaman, L. Rutkowski, R. Buyya, Task offloading strategies for mobile edge computing: A survey, Comput. Netw. 254 (2024) 110791, http://dx.doi.org/10.1016/j.comnet.2024.110791, URL https://www.sciencedirect.com/science/article/pii/S1389128624006236.

[18] G. Liu, H. Shi, A. Kiani, A. Khreishah, J. Lee, N. Ansari, C. Liu, M.M. Yousef, Smart traffic monitoring system using computer vision and edge computing, IEEE Trans. Intell. Transp. Syst. 23 (8) (2022) 12027–12038, http://dx.doi.org/10.1109/TITS.2021.3109481.

[19] I. Labriji, F. Meneghello, D. Cecchinato, S. Sesia, E. Perraud, E.C. Strinati, M. Rossi, Mobility aware and dynamic migration of MEC services for the internet of vehicles, IEEE Trans. Netw. Serv. Manag. 18 (1) (2021) 570–584, http://dx.doi.org/10.1109/TNSM.2021.3052808.

[20] L. Bréhon-Grataloup, R. Kacimi, A.-L. Beylot, Mobile edge computing for V2X architectures and applications: A survey, Comput. Netw. 206 (2022) 108797, http://dx.doi.org/10.1016/j.comnet.2022.108797, URL https://www.sciencedirect.com/science/article/pii/S1389128622000263.

[21] S. Garg, A. Singh, S. Batra, N. Kumar, L.T. Yang, UAV-empowered edge computing environment for cyber-threat detection in smart vehicles, IEEE Netw. 32 (3) (2018) 42–51, http://dx.doi.org/10.1109/MNET.2018.1700286.

[22] S. Niknam, A. Roy, H.S. Dhillon, S. Singh, R. Banerji, J.H. Reed, N. Saxena, S. Yoon, Intelligent O-RAN for beyond 5G and 6G wireless networks, in: 2022 IEEE Globecom Workshops, GC Wkshps, 2022, pp. 215–220, http://dx.doi.org/10.1109/GCWkshps56602.2022.10008676.

[23] E. Simanjuntak, N. Surantha, S.M. Isa, Evaluation of time-series database on microservice architecture for health monitoring system, in: 2022 International Symposium on Electronics and Smart Devices, ISESD, 2022, pp. 1–6, http://dx.doi.org/10.1109/ISESD56103.2022.9980618.

[24] M. Tsampazi, S.D. Oro, M. Polese, L. Bonati, G. Poitau, M. Healy, M. Alavirad, T. Melodia, PandORA: Automated design and comprehensive evaluation of deep reinforcement learning agents for open RAN, IEEE Trans. Mob. Comput. (2024) 1–18, http://dx.doi.org/10.1109/TMC.2024.3505781.

[25] L. Bonati, S. D'Oro, M. Polese, S. Basagni, T. Melodia, Intelligence and learning in O-RAN for data-driven NextG cellular networks, IEEE Commun. Mag. 59 (10) (2021) 21–27, http://dx.doi.org/10.1109/MCOM.101.2001120.

[26] A. Lacava, M. Polese, R. Sivaraj, R. Soundrarajan, B.S. Bhati, T. Singh, T. Zugno, F. Cuomo, T. Melodia, Programmable and customized intelligence for traffic steering in 5G networks using open RAN architectures, IEEE Trans. Mob. Comput. 23 (4) (2024) 2882–2897, http://dx.doi.org/10.1109/TMC.2023.3266642.

[27] L.M. Meruje Ferreira, F. Coelho, J. Pereira, Databases in edge and fog environments: A survey, ACM Comput. Surv. 56 (11) (2024) http://dx.doi.org/10.1145/3666001.

[28] I. Pelle, M. Szalay, J. Czentye, B. Sonkoly, L. Toka, Cost and latency optimized edge computing platform, Electronics 11 (4) (2022) http://dx.doi.org/10.3390/electronics11040561, URL https://www.mdpi.com/2079-9292/11/4/561.

[29] R. Laigner, Y. Zhou, M.A.V. Salles, A distributed database system for event-based microservices, in: Proceedings of the 15th ACM International Conference on Distributed and Event-Based Systems, DEBS '21, Association for Computing Machinery, New York, NY, USA, 2021, pp. 25–30, http://dx.doi.org/10.1145/3465480.3466919.

[30] A. Furda, C. Fidge, O. Zimmermann, W. Kelly, A. Barros, Migrating enterprise legacy source code to microservices: On multitenancy, statefulness, and data consistency, IEEE Softw. 35 (3) (2018) 63–72, http://dx.doi.org/10.1109/MS.2017.440134612.

[31] U. Kulkarni, A. Sheoran, S. Fahmy, The cost of stateless network functions in 5G, in: Proceedings of the Symposium on Architectures for Networking and Communications Systems, ANCS '21, Association for Computing Machinery, New York, NY, USA, 2022, pp. 73–79, http://dx.doi.org/10.1145/3493425.3502749.

[32] O-RAN Alliance, O-RAN whitepaper - building the next generation RAN, 2018, https://www.o-ran.org/resources.

[33] M. Polese, L. Bonati, S. D'Oro, S. Basagni, T. Melodia, Understanding O-RAN: Architecture, interfaces, algorithms, security, and research challenges, IEEE Commun. Surv. Tutor. 25 (2) (2023) 1376–1411, http://dx.doi.org/10.1109/COMST.2023.3239220.

[34] A. Calagna, Y. Yu, P. Giaccone, C.F. Chiasserini, Design, modeling, and implementation of robust migration of stateful edge microservices, IEEE Trans. Netw. Serv. Manag. (2023) http://dx.doi.org/10.1109/TNSM.2023.3331750.

[35] S. Gilbert, N.A. Lynch, Perspectives on the CAP theorem, Computer 45 (02) (2012) 30–36, http://dx.doi.org/10.1109/MC.2011.389.

[36] L. Larsson, W. Tärneberg, C. Klein, E. Elmroth, M. Kihl, Impact of etcd deployment on Kubernetes, Istio, and application performance, Softw.: Pr. Exp. 50 (10) (2020) 1986–2007, http://dx.doi.org/10.1002/spe.2885.

[37] Y. Wang, W. Gong, H. Fang, An NVMe-oF distributed storage design based on Etcd, in: 2022 4th International Conference on Frontiers Technology of Information and Computer, ICFTIC, 2022, pp. 701–705, http://dx.doi.org/10.1109/ICFTIC57696.2022.10075110.

[38] S. Chen, X. Tang, H. Wang, H. Zhao, M. Guo, Towards scalable and reliable in-memory storage system: A case study with redis, in: 2016 IEEE Trustcom/BigDataSE/ISPA, 2016, pp. 1660–1667, http://dx.doi.org/10.1109/TrustCom.2016.0255.

[39] G. Muradova, M. Hematyar, J. Jamalova, Advantages of Redis in-memory database to efficiently search for healthcare medical supplies using geospatial data, in: 2022 IEEE 16th International Conference on Application of Information and Communication Technologies, AICT, 2022, pp. 1–5, http://dx.doi.org/10.1109/AICT55583.2022.10013544.

[40] D. Ongaro, J. Ousterhout, Raft: In search of an understandable consensus algorithm, in: Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference, in: USENIX ATC'14, USENIX Association, USA, 2014, pp. 305–320.

[41] M. Iorio, A. Palesandro, F. Risso, Crownlabs—A collaborative environment to deliver remote computing laboratories, IEEE Access 8 (2020) 126428–126442, http://dx.doi.org/10.1109/ACCESS.2020.3007961.

[42] Prometheus, Open-source systems monitoring and alerting toolkit, 2015-2024, https://prometheus.io and https://github.com/prometheus/prometheus.

[43] Kepler, Kubernetes-based efficient power level exporter, 2015-2024, https://sustainable-computing.io/ and https://github.com/sustainable-computing-io/kepler.

[44] Cloud Native Computing Foundation (CNCF): Environmental Sustainability, Idle power matters: Kepler metrics for public cloud energy efficiency, 2024, https://tag-env-sustainability.cncf.io/blog/2024-06-idle-power-matters-kepler-metrics-for-public-cloud-energy-efficiency/.

[45] M. Amaral, H. Chen, T. Chiba, R. Nakazawa, S. Choochotkaew, E.K. Lee, T. Eilam, Kepler: A framework to calculate the energy consumption of containerized applications, in: 2023 IEEE 16th International Conference on Cloud Computing, CLOUD, 2023, pp. 69–71, http://dx.doi.org/10.1109/CLOUD60044.2023.00017.

[46] C. Centofanti, J. Santos, V. Gudepu, K. Kondepu, Impact of power consumption in containerized clouds: A comprehensive analysis of open-source power measurement tools, Comput. Netw. 245 (2024) 110371, http://dx.doi.org/10.1016/j.comnet.2024.110371, URL https://www.sciencedirect.com/science/article/pii/S1389128624002032.

[47] M. Akbari, R. Bolla, R. Bruschi, F. Davoli, C. Lombardo, B. Siccardi, A monitoring, observability and analytics framework to improve the sustainability of B5G technologies, in: 2024 IEEE International Conference on Communications Workshops, ICC Workshops, 2024, pp. 969–975, http://dx.doi.org/10.1109/ICCWorkshops59551.2024.10615948.

[48] Bitnami, Etcd helm chart, 2022-2024, https://artifacthub.io/packages/helm/bitnami/etcd.

[49] Bitnami, Redis helm chart, 2022-2024, https://artifacthub.io/packages/helm/bitnami/redis and https://artifacthub.io/packages/helm/bitnami/redis-cluster.