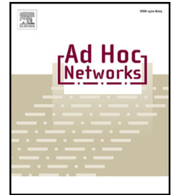
Ad Hoc Networks 156 (2024) 103413

Contents lists available at ScienceDirect

Ad Hoc Networks

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



CACHE-IT: A distributed architecture for proactive edge caching in heterogeneous IoT scenarios

Ivan Zyrianoff [a,b,*], Lorenzo Gigli [a,b], Federico Montori [a,b], Luca Sciullo [a,b], Carlos Kamienski [c], Marco Di Felice [a,b]

[a] Department of Computer Science and Engineering, University of Bologna, Italy
[b] Advanced Research Center on Electronic Systems "Ercole De Castro", University of Bologna, Italy
[c] Federal University of ABC (UFABC), Santo André, SP, Brazil

ARTICLE INFO

Keywords:
Internet of Things
Edge computing
Proactive caching
Cloud-to-Thing continuum
Edge caching
Caching architecture

ABSTRACT

The Cloud-to-Things (C2T) continuum combines the proximity of edge infrastructure to the devices with cloud resources to optimize data processing and response time in the Internet of Things (IoT). Proactive edge caching is a potential solution for meeting latency constraints in C2T environments, enabling efficient data processing and storage while reducing redundant computation and cost. However, while 5G/6G infrastructural aspects and caching strategies are extensively studied, no frameworks facilitate the design and deployment of caching strategies or address IoT's unique requirements. This paper proposes CACHE-IT, a proactive edge caching framework that decouples the caching strategy algorithm from the underlying architecture, enabling customization based on application-specific requirements. Through extensive simulations, we analyzed the impact of different configurations on system metrics and verified that the CACHE-IT positively impacts the system latency and hit rate. By implementing a scenario-specific caching strategy, we illustrate the CACHE-IT deployment in a real-world Structure Health Monitoring (SHM) system. The evaluation demonstrates that CACHE-IT impacts positively in terms of latency, hit rate, and the number of requests sent to data providers.

1. Introduction

In recent years, we experienced a rapid growth of low-power devices connected to the Internet, which brought forth a plethora of new challenges for computational and network infrastructure. The Internet of Things (IoT) has become a primary enabler for various new domains, such as Smart Cities [1] and Structure Health Monitoring (SHM) [2]. However, IoT devices have constrained processing capabilities and struggle to handle the increasing demands of data processing and storage. Consequently, cloud computing was integrated with IoT, as it offers virtually unlimited storage and computation capabilities to the thousands of gigabytes daily produced by sensory data [3]. Nevertheless, cloud computing faces its own set of limitations regarding IoT. In particular, the long communication delay between users and the cloud hinders IoT time-sensitive applications that rely on high automation and low-latency operations, such as in Industry 4.0 [4,5] and vehicle networks [6]. Edge computing has gained prominence as a potential solution to overcome these challenges by bringing data and computation near IoT devices. By leveraging the client proximity of edge devices combined with the vast resources of the cloud, the novel Cloud-to-Things (C2T) continuum paradigm [7] aims to optimize data processing and storage, providing a balance between latency-sensitive applications and resource-intensive cloud computations.

Not all computational tasks can be efficiently offloaded to the network edge due to constraints in the processing power and robustness of edge nodes [8]. Further, real IoT systems commonly require to interface with third-party services that are not under system administrators' control [9]. In order to achieve better exploitation of C2T integration with IoT, edge caching constitutes a potential solution to satisfying latency constraints [10]. Storing frequently accessed data at the edge reduces redundant computation, which leads to cost savings in terms of power and cloud resources — e.g., serverless functions, cloud data transfer, and paid APIs. More in-depth, proactive edge caching [11–13] is attracting attention as a further improvement of edge caching. Its principle is to explore the relationship between users and their request patterns and accurately predict and prefetch data, reducing latency while satisfying freshness of information constraints [14] — i.e., Age of Information (AoI). However, the current literature has two gaps that need to be addressed.

* Corresponding author at: Department of Computer Science and Engineering, University of Bologna, Italy.
E-mail addresses: ivandimitry.ribeiro@unibo.it (I. Zyrianoff), lorenzo.gigli@unibo.it (L. Gigli), federico.montori2@unibo.it (F. Montori), luca.sciullo@unibo.it (L. Sciullo), cak@ufabc.edu.br (C. Kamienski), marco.difelice3@unibo.it (M. Di Felice).

https://doi.org/10.1016/j.adhoc.2024.103413
Received 5 June 2023; Received in revised form 11 October 2023; Accepted 18 January 2024
Available online 1 February 2024
1570-8705/© 2024 The Author(s). Published by Elsevier B.V. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

First, the studies focus mainly on the optimization of edge caching in scenarios such as 5G and 6G networks, where the primary goal is to minimize backhaul traffic and optimize network performance from the perspective of network owners [15], which do not overlap with the requirements imposed by IoT scenarios. Apart from the time requirements of IoT applications, the diverse range of IoT devices and protocols demand interoperability to enable seamless integration and communication [16]. IoT environments are highly dynamic, characterized by rapidly changing conditions through continuous device addition and removal, and their mobility [17]. Unlike network-oriented proactive caching, which focuses on ensuring average client satisfaction, IoT application-oriented caching has varying constraints. For instance, in vehicular networks, the infotainment data system needs low latency sufficient to meet clients' Quality of Experience (QoE), while essential navigation data requires strict AoI and latency [18]. These factors present significant challenges demanding a tailored proactive edge caching solution designed explicitly for IoT environments.

Second, caching strategies, models, and optimizations are the main focus of researchers. At the same time, there is a lack of proposals for caching frameworks that enable the design and deployment of caching strategies [19]. The absence of design efforts to guide users on the implementation and deployment creates a significant gap between academia and practitioners; while complex caching strategies are consolidated in literature, their implementation requires time-demanding efforts from highly specialized individuals.

To address these challenges, we propose CACHE-IT (**C**onnected **A**rchitecture for **C**aching **HE**terogeneous **IoT**), a distributed framework for proactive edge caching in IoT scenarios. CACHE-IT allows rapid deployment, modification, and replacement of the caching strategy. The framework provides the clients' request history as input to the caching strategy, which is handled as an independent and modular component. CACHE-IT components are based on current Web technology standards, ensuring compatibility and easy integration with existing systems.

The contributions of our paper are:

- **Flexibility**: CACHE-IT allows the deployment of customized and flexible caching in the C2T continuum by decoupling the edge caching architecture from the caching strategy, enabling use-case-driven caching strategy customization. Consequently, it provides versatility in the technique utilized (e.g., Deep Learning, Heuristic) and the caching goal (e.g., minimize latency, minimize AoI). The framework allows handling different specific users or requests that impose specific constraints.
- **IoT oriented design**: we address interoperability issues of IoT by providing a dedicated device abstraction layer that seamlessly integrates heterogeneous IoT devices through a standard and well-defined interface. We tackle the dynamism of IoT environments regarding the volatile nature of sensors by incorporating mechanisms that adapt the caching strategy to reflect these changes. Finally, CACHE-IT components were designed to be distributed in the C2T continuum, considering edge nodes ranging from base stations to constrained computational devices.
- **Advanced Caching Mechanism**: CACHE-IT utilizes cooperative and proactive caching mechanisms for increased performance with no additional complexity. It optimizes the capacity of the system to share resources in a twofold manner. First, caching strategies are able to redirect specific Cache Node requests to query directly other edge nodes. Second, when a cache miss occurs, the Cache Node queries its nearest neighbors. We intentionally avoided mechanisms that might be bound to specific caching strategies to ensure CACHE-IT flexibility and adaptability.
- **Twofold Validation**: we performed extensive simulations varying numerous parameters to understand the impact of CACHE-IT features under different scenarios and client behaviors — modeled according to patterns commonly found in IoT scenarios. Additionally, we deployed CACHE-IT in a real-world Structure Health Monitoring (SHM) system, which illustrates how CACHE-IT can provide personalized strategies to specific IoT environments. We evaluated the testbed by comparing the system with and without traditional caching mechanisms.

In the remainder of this paper, Section 2 discusses related works in the field and compares them to our solution. Section 3 introduces the CACHE-IT framework, providing a comprehensive overview of its architecture, while Section 4 details its operations. Section 5 describes CACHE-IT implementation. In Section 6, we evaluate the framework performance through large-scale simulations while CACHE-IT deployment in a real IoT environment is described in Section 7. Finally, Section 8 concludes the paper and discusses future research directions that have emerged from this work.

2. Related work

The topic of edge caching has been deeply explored in literature. For a more structured reading, we redirect the interested reader to our previous work, where we proposed a taxonomy for IoT edge caching solutions [19]. In this section, we review the recent literature and highlight the differences concerning our work by focusing on the following aspects:

- The **goal** of the caching strategy, i.e., what the caching strategy aims to optimize.
- The overall caching **strategy**, focusing on the underlying scientific method adopted.
- Whether the caching solution is **domain-agnostic** or is particularly tied to one application, e.g., video streaming.
- Whether the caching solution is **infrastructure-agnostic** or is particularly tied to a certain physical deployment, e.g., 5G macro-cells.
- Whether the caching solution is **proactive**, i.e., it tries to cache the content before it is requested.
- Whether the caching solution is **cooperative**, meaning that the edge Cache Nodes exchange content or information for tuning their decisions.

Related information about each of the analyzed papers is collected and summarized in Table 1, while the rest of the section is dedicated to their description and the analysis of the gaps our work aims to tackle.

First, a substantial quantity of works proposed in the literature are bound to a definite application scenario or deployment. It is the example of [4], which deals specifically with industrial scenarios where mobile nodes need data from a central server and wander near wireless sensors. Here, the authors propose a centralized optimization algorithm where all the computation takes place in the cloud to proactively offload data to edge servers by predicting client mobility. Prediction of mobility patterns is also employed in [20], where proactive caching is used in urban scenarios to deliver content to vehicles promptly. In [21], authors cache and transcode video content on UAVs to alleviate the backhaul load through a heuristic model. Authors in [22] propose a deep learning model for proactive caching that predicts the content popularity of videos and music only. In contrast with our work, the mentioned approaches are vertical in one application context and lack adaptability.

One recurrent scenario for which a consistent number of works have been proposed is 5G cellular networks. In [23], authors overview the state-of-the-art and the current challenges and potential solutions for 5G edge caching. They also propose a caching framework that leverages blockchain transactions and non-negative matrix factorization. In [24], authors inspect cooperative micro-caching as a network function that should be embedded in 5G networks to minimize latency. A similar 5G scenario is analyzed in [6], where results aim to motivate the introduction of new caching policies to deal with high mobility nodes

Table 1
Comparison of CACHE-IT with the works in literature.

Paper	Goal	Strategy	Domain	Infrastructure	Proactive	Cooperative
[12]	Backhaul Load	Deep Learning	✓	✗	✓	✗
[11]	Hit Rate	Deep Learning	✗	✗	✓	✗
[28]	Delay	No strategy	✓	✓	✗	✗
[31]	Delay, Hit rate	Deep Learning	✓	✓	✓	✓
[29]	Response Time	Heuristic	✗	✓	✗	✓
[32]	Hit Rate	Deep R. Learning	✓	✓	✗	✓
[4]	Delay, Goodput	Optimization	✗	✗	✓	✗
[24]	Delay	Fuzzy inference	✓	✗	✗	✓
[6]	Delay	Statistical	✓	✗	✗	✗
[13]	Hit Rate	LSTM & Ensemble	✓	✗	✓	✗
[25]	Hit Rate & Delay	Deep R. Learning	✓	✗	✓	✓
[20]	Hit Rate & Delay	Probabilistic	✗	✗	✓	✗
[21]	Hit Rate & Delay	Heuristic	✗	✗	✗	✗
[35]	Hit Rate & Delay	LSTM	✓	✓	✓	✓
[33]	Hit Rate & Profit	Federated Learning	✓	✓	✓	✓
[34]	Traffic Load	Deep Learning	✗	✗	✓	✗
[36]	Hit Rate, Delay	Heuristic	✓	✗	✗	✗
[30]	User Utility	Probabilistic	✗	✗	✗	✓
[23]	Delay	Optimization	✓	✗	✓	✗
[26]	Hit Rate	Deep R. Learning	✓	✗	✓	✓
[22]	Hit Rate, Load	Deep Learning	✗	✗	✓	✗
[27]	Traffic Load	Greedy	✓	✗	✗	✓
CACHE-IT	Flexible	Flexible	✓	✓	✓	✓

(e.g., vehicles). Again, in [25], authors focus on mobile content caching for 5G networks, tackling the problems of edge caching and radio resource allocation separately, using a deep reinforcement leaning for the first one and a branch & bound approximation algorithm for the second. The work in [12] proposes DeepCacheNet, a deep learning-based framework for proactive caching that uses stacked denoising autoencoders to classify content popularity and instruct base stations to cache such content. In [26], the authors present a framework for proactive and cooperative edge caching that aims to cluster base stations to establish intra-cluster collaboration through a deep reinforcement learning technique. Caching-as-a-Service [27] is a caching virtualization framework along with the development of Cloud-based Radio Access Networks (C-RAN). It focuses on virtualizing the cache through Network Function Virtualization (NFV) and exploring the possibilities of detaching the software application from the underlying hardware. These approaches, unlike ours, are tied to the 5G infrastructure, requirements, and characteristics, which do not reflect many IoT systems.

Most edge caching works in literature present a similar core structure: they aim to optimize several parameters (e.g., the hit rate and the backhaul load) of a specific caching policy. Therefore, they propose a mathematical structure to model the system, an algorithmic or learning-based solution, and numerical results supported by simulations. Some solutions are numerical or probabilistic, such as Smart-Edge-CoCaCo [28], or [29], which proposes a cooperative caching mechanism for scenarios where edge nodes offload computing tasks to edge cloudlets. A D2D solution is adopted in [30], where authors consider data freshness and client mobility. However, most of the solutions in the literature rely on deep learning, as [11], where content popularity in the future is predicted by using bidirectional deep recurrent neural networks, or [31], which uses a distributed deep learning algorithm to predict the content demand by single users, or in [32], where deep reinforcement learning is adopted by each caching agent so that they learn to cooperate and share cached resources. Other related deep learning-based works use federated learning [33] or deep learning in conjunction with attention mechanisms [34]. In [35], authors present a very different solution: proactive caching is done locally in each user's equipment, and cache hit can occur locally or with direct D2D communication, eliminating the structural constraints enforced by cellular networks. With such a focus on the single caching strategy, very few efforts are dedicated to proposing an edge caching architecture for IoT scenarios capable of accommodating different caching policies. Furthermore, most of the solutions proposed are not complemented by a real implementation or guidelines of the software components.

It is also important to mention Information-Centric Networking (ICN), which, in the past decade, has established itself as an alternative to conventional TCP/IP networks. ICN, in particular in its guise of Named Data Networking (NDN), fulfills a content-centric design and a location-independent naming mechanism, giving this paradigm several advantages in the scope of mobility and efficiency. One powerful and native feature of ICN is its in-network caching in intermediate network routers [37]. The recent study in [36] presents an all-encompassing solution based on NDN, encompassing network topology, data freshness, and content popularity, thus attempting to design a common solution to all previous works on caching in ICN.

A final aspect to consider is modeling the pattern of requests from clients. A meaningful amount of the mentioned works assume time-invariant content popularity and primarily cater to human clients. This may differ if the system actors are IoT devices, such as sensors or robots. Furthermore, they disregard the system aspects of how to collect, share, store, and process client traces to obtain the popularity values of each resource. In [13], authors apply proactive caching to mobile edge networks at the base stations based on content popularity; however, they evaluate the method over a movie dataset, assuming to know the demographic information of the clients through the usage of cameras. In contrast to existing approaches, our work focuses on crucial aspects of IoT environments, including time-variant conditions and interoperability. Additionally, we evaluated CACHE-IT under different client behaviors, which was modeled based on the real behavior of agents in IoT scenarios.

3. CACHE-IT architectural design

CACHE-IT is an architectural framework for proactive edge caching in heterogeneous IoT scenarios that provides high customization and flexibility. Following, we describe the guidelines that support CACHE-IT design (Section 3.1), then we detail its high-level architecture (Section 3.2).

3.1. Designing guidelines

Our architecture aims to provide proactive caching capabilities through a fully customized framework. Additionally, our solution aims to be deployed on top of operational IoT systems. Four general guidelines supported our architectural design:

1. **Interoperability**: we adopt standards-based approaches and open interface solutions that enable seamless communication between heterogeneous devices and data providers.
2. **Generality**: IoT systems are employed in a wide range of domains, each with unique requirements and characteristics. These domains include areas that are orthogonally different from each other, such as Industry 4.0 and smart agriculture. Specific scenarios may have their own particularities that must be addressed. The architecture should be domain-agnostic and capable of handling different constraints.
3. **Adoption**: The fundamental elements of the architecture must rely on established, real-world technologies that are widely adopted in the industry. Nonetheless, the architecture description must remain independent of the underlying technology
4. **Flexibility**: The architecture must be flexible regarding the caching deployment in the C2T continuum by decoupling it from the caching strategy regarding its goal, constraints, and requirements. The ability to easily switch between caching policies allows organizations to react rapidly to alterations in their requirements or constraints.

3.2. CACHE-IT high level architecture

CACHE-IT is a distributed microservice-oriented architecture; each software component is modular and independent. This design addresses scalability concerns by allowing multiple replicas of the same services to be instantiated simultaneously, effectively managing a high demand of requests to the framework. Fig. 1 illustrates CACHE-IT high-level architecture, which comprises clients – at the bottom – that request data from providers — at the top. Requests are routed through the caching framework, which checks if the requested content is cached and valid. If so, data is returned to the client without forwarding the request to the provider.

CACHE-IT comprises two classes of computational nodes: a single Cache Controller and N_C Cache Nodes. The set of Cache Nodes is denoted as $C = \{c_1, c_2, \dots, c_{N_C}\}$. The Cache Manager – a Cache Controller component – generates a set of instructions, called caching orders, for each Cache Node that determine: *what*, *when*, and *how long* to cache. The Cache Workers – one per Cache Node – are responsible for carrying out the caching orders – i.e., performing requests and storing data – and managing clients' requests, thus returning the cached content or forwarding their request to the specified provider. Typically, Cache Nodes are located at the edge, and the Cache Controller is deployed in the cloud. Any device with an operating system and storage capabilities is suitable as a Cache Node. It can be deployed in a dedicated configuration, such as on a Raspberry Pi, or as an independent and isolated process within a shared environment. The latter setup is better suited for more powerful and stable equipment (e.g., base stations). However, we highlight that the Cache Node does not engage in resource-intensive processing tasks (which are executed by the Cache Manager), resulting in minimal impact on the node's computational metrics. A caching order is a well-defined structure that contains an instruction to be performed by the specified Cache Node at a certain execution time, and its output is valid until the specified expiration time. A caching order has two different modes:

- *Standalone*: the caching order contains the provider to be requested and all the metadata needed to formulate the request. Each cache order makes the Cache Worker perform a request to a given provider; the corresponding response is cached for a predetermined time.
- *Cooperative*: besides the content in standard mode, the caching order includes an additional parameter: a Cache Node address. In this case, instead of requesting data directly to the service provider, the Cache Worker checks if the Cache Node specified has the requested resource; if so, the requests to the specified provider will be fetched from that Cache Node until the defined expiration time.

By employing cooperative caching orders, the Cache Manager can minimize the number of requests made to the providers. This is achieved by storing a particular resource in one Cache Node and directing the remaining Cache Nodes to retrieve it from that specific Cache Node.

Before initialization, the user defines the *Caching Template*, a well-defined structure that holds all the system configurations — including the caching strategy. Table 2 lists the attributes included in the Caching Template along with a brief description. There are four independent data flows in CACHE-IT (as depicted in Fig. 1 legend), and for each Caching Template attribute, we denote the data flow it impacts and the subsection that describes it.

4. CACHE-IT operations

After initialization, CACHE-IT has a short bootstrap phase for enabling interoperability (Section 4.1), followed by the caching strategy execution (Section 4.3), which is also triggered every t time slots (the time slot duration set in the Caching Template). Both the history manager (Section 4.2), the request forward, and data retrieval data flows (Section 4.4) are triggered by the arrival of the client's requests.

4.1. Interoperability enabler

An important aspect to consider when designing an IoT caching framework is the heterogeneous nature of the devices, which vary greatly in software and communication protocols. As illustrated by Fig. 1, CACHE-IT clients can be diverse, ranging from constrained sensors to robots and even regular humans. IoT devices do not usually adopt compatible interfaces with current Web technologies, which makes integration with non-IoT solutions difficult. To address this issue, CACHE-IT provides a standardized and interoperable interface, allowing clients to abstract the specific interface adopted by the data providers.

CACHE-IT bootstrap operation aims to enable interoperability, as illustrated in red in Fig. 1. The active component in this data flow is the Interface Translator, which bridges dissonant ecosystems, namely the traditional Web, with a standard and interoperable interface for IoT devices. We adopt the W3C Web of Things (WoT) [38] standard, as it extends existing Web technologies and provides a homogeneous interface to access IoT devices that abstract from their particular interfaces and heterogeneous network protocols. The WoT core component is the Web Thing (WT), an abstraction of a physical or a virtual entity with its metadata and interfaces described in a well-defined document called Thing Description (TD). The TD describes the WT metadata, capabilities, properties, and interactions in a machine- and human-understandable structure — i.e., JSON. The interaction with a given WT is implemented through the WT protocol bindings, which define the mapping of the possible device interactions to different network protocols.

As WoT does not provide methods or guidelines to convert dissonant interfaces to its ecosystem, the Interface Translator converts traditional Web services interfaces (i.e., RESTful APIs) into TDs and instantiates them in WTs that act as a proxy of the original provider. *Step R1* in Fig. 1 refers to the syntactical translation of the provider interface into a TD, and *Step R2* depicts its instantiation into a proxy WT. The set of data providers to translate are defined by the `providers` attribute of the Caching Template.

4.2. History management

CACHE-IT defines a pipeline to manage the clients' request history, which the caching strategies may use. Each request performed by a client – through the interoperability layer – is registered in a `log` file, as depicted in the *Step G1* of the dataflow. The lightweight nature of `log` files makes them well-suited for resource-constrained IoT environments. Additionally, the system also supports a configurable

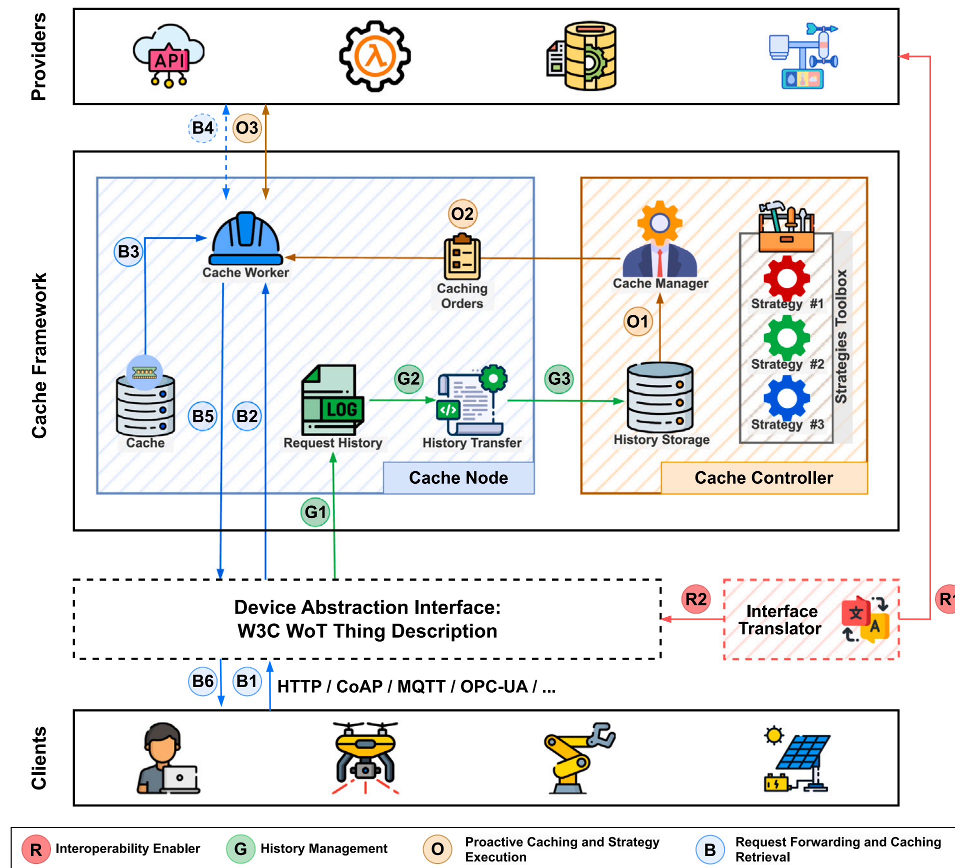


Fig. 1. CACHE-IT High Level Architecture.

Table 2
Caching Template.

Attribute	Description	Data flow	Subsection
`providers`	data providers to be translated to a seamless interface	interoperability	4.1
`record attributes`	what attributes to register from clients requests	history management	4.2
`logs longevity`	retention time of log files	history management	4.2
`t time slot duration`	the time slot in which a new set of caching orders is generated	caching strategy execution	4.3
`caching strategy`	a set of three functions that generate caching orders	caching strategy execution	4.3
`cache node storage`	the maximum storage available in each cache node	caching strategy execution	4.3
`cache replacement`	resource replacement strategy	request forwarding and data retrieval	4.4
`cNN`	the number of neighbor Cache Nodes visited if the resource is found locally	request forwarding and data retrieval	4.4
`reactive caching`	if the data resulting from a cache miss is stored	request forwarding and data retrieval	4.4

Table 3
Properties of a record.

Property	Description	Optional
`request`	IP Address or DNS, port, and URI	✗
`parameters`	query, path, header, and cookie parameters	✗
`cached`	boolean value that is true if the requested content was retrieved from the cache	✗
`delay`	time elapse to retrieve the response (from cache or the response time from performing the request to the provider)	✗
`timestamp`	request arrival timestamp	✗
`cache node`	the Cache Node received and registered the record	✗
`client`	client identification (IP and MAC address)	✓
`data`	returned request data	✓

data-pruning mechanism to ensure the efficient management of the `log` files. This functionality allows the user to define record retention parameters, namely `logs longevity` defined in the Caching Template. Logs surpassing a pre-set age threshold are automatically pruned based on these parameters, ensuring the lightweight nature of the `log` files over an extended duration.

Each record's core information is its arrival time (i.e., its timestamp) and the Cache Node where the record was registered. The complete content of a record is depicted in Table 3. Some record properties are optional. Namely, the returned data, which might be too burdensome to store, manage, and process, and the client identification, which might be concealed for privacy reasons. The user defines through the Caching Template which optional properties will be registered; this choice is based on the information and metadata that the employed caching strategy utilizes.

Step G2 and *Step G3* refer to transferring the request history of a single Cache Node to the centralized Cache Controller storage, a process that occurs in batches. The History Transfer is a lightweight shipper for forwarding and centralizing `log` data. In detail, *Step G2* illustrates the operation of finding and managing all the `log` files that will be transferred to the Cache Controller. The History Transfer needs to keep track of the state of each `log` file and what portion of the file was not sent already. *Step G3* refers to the periodical data transfer of the collected request history to the centralized storage in the Cache Controller. We define as D_{past} the subset of data points stored in the last $past$ time. The History Transfer needs to attend to some requirements:

- **Disconnection Handling**: as the Cache Nodes often are unreliable computation nodes in the network edge, they might suffer occasional disconnections from the Internet; hence, they lose communication with the Cache Controller. In such cases, the History Transfer should keep track of the unsent batches and send them once the connection is re-established.
- **Lightweight implementation**: the History Transfer should use limited system resources and not impact the other process executing in the Cache Node.
- **Multi-environment deployment**: there are virtually no constraints on what devices can fulfill the role of a Cache Node. The History Transfer should be able to support multi-heterogeneous deployment environments.
- **History Update Management**: as common scenarios comprise multiple Cache Nodes, those can potentially overwhelm the History Storage. The History Transfer must employ techniques to find an adequate pace to transmit data to the History Storage.

4.3. Caching strategy execution

The Cache Manager comprises three distinct operations, each with different execution periodicity and detailed in its own Subsection. The modification and deployment of a Caching Template occur upon direct user intervention, and it is a *long-term* operation. Adapting the caching strategy to new environmental conditions is a *medium-term* operation. Lastly, the generation and transmission of caching orders are considered a *short-term* operation. Fig. 2 depicts each operation and its interactions. Additionally, `CACHE-IT` *short-term* operations are depicted through the orange dataflow in Fig. 1. Although they are explained in detail in Section 4.3.3, an introduction is necessary to understand the big picture, as these operations are a core system component: every t time slot (the duration of which is specified in the Caching Template) the Cache Manager – located in the Cache Controller – invokes a function that generates and transmits (*Step O1* and *Step O2*) instructions (i.e., caching orders) to each Cache Worker. We count the time slots as $t_0, t_1, \ldots, t_n$. When a Cache Worker receives a new batch of caching orders, it reports its caching accuracy in the previous time slot (i.e., t_{i-1}) to potentially trigger *medium-term* operations that adjust the generation of such caching orders (such as retraining a model), meaning that the conditions may have changed (see Section 4.3.2 for details).

4.3.1. Long-term: user intervention

Before system initialization, the user defines the caching strategy in the Caching Template. In proactive edge caching, the strategies can employ diverse techniques that impose different data flows and operations [39]. For instance, a statistical-based strategy requires re-execution for generating each new set of caching orders, while a machine learning-based strategy relies on a pre-trained model, which requires an understanding of the timing for training (and retraining) the model. To encompass variability in the strategies processing steps, we define a generic workflow comprising three functions that compose the caching strategy in `CACHE-IT`. Those functions are:

- gen **function**: the core function that sets the strategy goal and implements a technique (e.g., Deep Learning). It is a higher-order function that produces a function $cOrder$, which generates a set of caching orders in each time slot (*short-term* operation). The gen function is analogous with the machine learning process of *training a model*, while the $cOrder$ function pairs with the model execution itself in inference mode. Other techniques that do not involve pre-computing steps can be integrated into the framework by defining gen as a deterministic function that always outputs the same $cOrder$ function.
- $period$ **function**: the function accesses the historical storage and outputs a time slot, determining the size of the data chunk utilized by the gen function upon execution. The aim is to determine the last homogeneous time segment of historical storage data. It is possible to utilize a simple rolling window method (e.g., always get the last 1-month of data) or an advanced technique (e.g., an auto-regression model that, based on the prediction errors, determines the last time segment of homogeneous data).
- $trigger$ **function**: it is executed every time slot (i.e., *short-term*), taking as input the accuracy of all the caching nodes in the last time window and returns a `Boolean` value. If `true`, it prompts the execution of the *period* and the *gen* functions to regenerate the $cOrder$. The output of the function can be bounded to the Cache Nodes accuracy (e.g., if less than 20%), time (e.g., every 12 h), or both.

Upon system initialization or when the user uploads a new Caching Template, `CACHE-IT` executes the newly defined $period$ and gen functions to generate an updated $cOrder$ function considering the newly set configurations, such as the maximum available storage for each Cache Node.

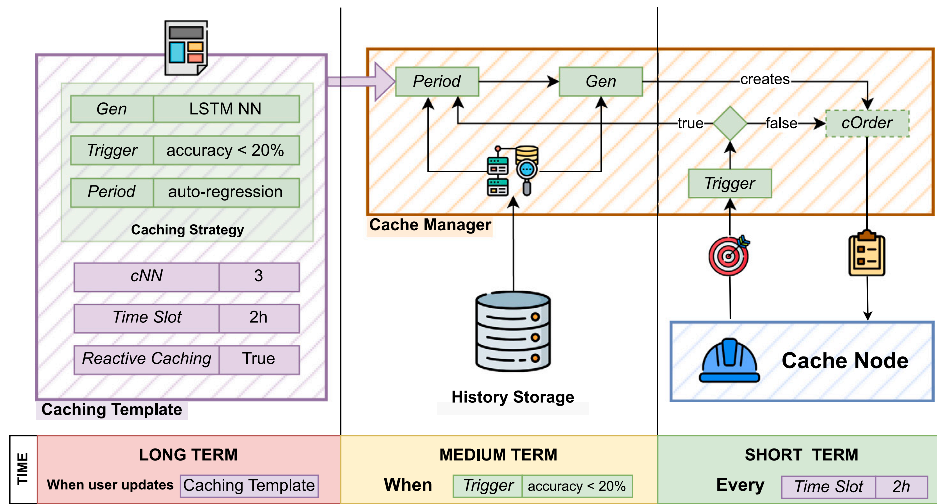


Fig. 2. CACHE-IT operations and their timings.

Fig. 1 omits the component in charge of managing the Cache Nodes and gathering data related to their status – i.e., online or offline – and their connectivity. However, we adopted IoTManA [40] as our chosen management system, which provides tools for managing, controlling, and monitoring software, hardware, and communication components. Specifically, it facilitates monitoring network delay and system entities' availability — i.e., the Cache Controller and Cache Nodes.

4.3.2. Medium-term: caching strategy adaption

Due to changes in the context, such as the inclusion, removal, or behavior change of IoT devices, a historical system snapshot may not represent its current state. To periodically adapt the generation of caching orders to reflect the system conditions better, CACHE-IT performs a check (e.g., executes $trigger$ function) each time slot to determine if conditions have changed. From CACHE-IT point-of-view, an alteration in system conditions means that the output of $trigger$ function is `true` or that a new Caching Template was uploaded. In such cases, CACHE-IT regenerates the $cOrder$ function by executing the $period$ function followed by the gen function.

4.3.3. Short-term: generation of caching orders

The *short-term* operations encompass the generation of caching orders in each time slot. Algorithm 1 is executed in fixed slots of t time to generate and transmit a new set of caching orders, detailing *Step O1* and *Step O2* from Fig. 1. Line 2 denotes the generated function $cOrder$ producing caching orders for the next t_{i+1} time slot. Then, the Cache Manager transmits to each Cache Worker its set of caching orders, which reply with their caching accuracy in the last t_{i-1} time slot (lines 3 to 5). A Cache Worker accuracy is defined as the total amount of cache hits divided by the total number of requests. The algorithm checks if gen is already in execution (line 6). Then, if the $trigger$ function outputs `true` (lines 7 and 8) the *medium-term* operations are triggered. Namely, the execution of $period$ (line 9), followed by gen (line 10); which updates the caching order generating function $cOrder$.

Fig. 2 illustrates an example of the three distinct caching operations. The upload of a new Caching Template (*long-term*) triggers the execution of $period$ and gen to generate a new $cOrder$ (*medium-term*). In the example, the gen function performs the retraining of an LSTM Neural Network model, which becomes the updated $cOrder$. Then, the Neural Network (i.e., $cOrder$) generates a new batch of caching orders in every time slot (e.g., 2 h), which are transmitted to each Cache Node (*short-term*). The Cache Nodes reply with their accuracy. The $trigger$ function then takes the set of accuracies as input and, if their average is below 20%, prompts the execution of the *medium-term* operations to retrain the Neural Network.

Algorithm 1: The CACHE-IT caching orders generation

Every t_i **time passed:**
2 $caching_orders_{t_i} \leftarrow cOrder(D_{past})$
3 **for** c *from* 0 **to** N_c **do**
4 $orders_c \leftarrow$ filter $caching_orders_{t_i}$ for caching orders that match c
5 transmit to the c Cache Node the $orders_c$ and receive $accuracy_{t_{i-1},c}$ of all Cache Workers
6 **if** *gen is not executing* **then**
7 $isTrigger \leftarrow trigger(\bigcup_{c \in C} accuracy_{t_{i-1},c})$
8 **if** *isTrigger* **then**
9 $past \leftarrow period(D)$
10 $cOrder \leftarrow gen(D_{past})$
11 **end**
12 **end**
13 **end**

4.4. Request forwarding and caching retrieval

The request forwarding and caching retrieval process employed by each Cache Node is triggered upon receiving a request; the overall flow is depicted in blue in Fig. 1. A client requests data from a given provider in any protocol supported by the Device Abstraction Interface, as depicted by *Step B1*, which translates the request and forwards it to the Cache Worker. In turn, the Cache Worker checks if the requested data is cached (*Step B3*). If so, the data is returned to the client (*Step B5* and *Step B6*). The role of *Step B6* is mapping the data to the format and protocol used by the client. When a cache miss occurs (*Step B4*), the Cache Worker forwards the request to the provider.

Algorithm 2 is executed by each Cache Worker to handle client data requests and to perform caching orders. One particular feature described in the Algorithm 2 is the cNN (CACHE-IT Nearest Neighbors) strategy. In cNN, when a cache miss occurs, the Cache Worker checks if its N nearest neighbors (i.e., other Cache Nodes) have the resource requested cached. This feature can be integrated into any caching strategy and capitalizes on the assumption that geographic proximity influences the request pattern. The amount of N of neighbors visited is defined in the Caching Template.

Algorithm 2 keeps track of the received caching orders, the local cache hit rate (i.e., its accuracy), and the set of N neighboring Cache Nodes and their latency towards the originating Cache Node. The Algorithm 2 uses two event handlers for dealing with the arrival of caching orders and client requests. For the sake of simplicity, we adopt

the dot notation (.) to access the properties of a caching order — e.g., $order.type$.

When a set of caching orders arrives (line 5), the algorithm checks if any client requests were made since the last batch of caching orders was received (line 6); if so, it computes the cache accuracy by dividing the number of cached requests by the total number and transmits it to the Cache Controller (lines 7-9). For each caching order in the set, the algorithm checks whether the order is of $standalone$ or $cooperative$ (lines 11-17). For standalone orders, the algorithm performs the request at the designated execution time and caches the response data until its expiration time; if the caching order is $cooperative$, the Cache Worker adds it to a list of cooperative orders (line 15). The list of cooperative orders is periodically updated to remove expired orders — omitted from the algorithm for clarity.

When a request arrives from a client (line 18), the algorithm first checks if the content is present in the local cache (line 20). If so, the corresponding data is returned to the client, and the $cachedRequests$ variable is incremented (line 21). If not, the Cache Worker checks if there is a valid cooperative caching order that matches the request (lines 22–27). If so, the Cache Worker tries to retrieve data from the Caching Node specified in the cooperative order, returning it to the client and incrementing $cachedRequests$ in case of success. If the data is not found, the algorithm performs the cNN strategy (lines 28–31) by verifying if the content requested is cached in any of the N nearest Cache Nodes. Finally, if data is not found, the algorithm performs the request directly to the data provider and returns the data to the client (lines 32–36). The returned resource is only cached if the variable $reactiveCaching$ is set to `true` This behavior considers the reliability of the prediction solution, as a cache miss may be an outlier, and caching its returned resource might be unnecessary.

CACHE-IT default cache replacement strategy is Least Recently Used (LRU). However, users can configure it to use other strategies such as Least Frequently Used (LFU), random, or maximum idle time.

Algorithm 2: The Cache Worker caching retrieval and request forwarding.

```
1  cooperativeOrders ← [ ]
2  cacheNodes ← list of CacheNodes and their respective latency
3  totalClientRequests ← 0
4  cachedRequests ← 0
5  upon the arrival of a set of caching orders orders: (every t
     intervals)
6      if totalClientRequests is not 0 then
7          accuracy ← cachedRequests/totalClientRequests
8      else
9          accuracy ← 1
10     end
11     transmit accuracy to Cache Controller and reset
        totalClientRequests and cachedRequests
12     for order in orders do
13         if order.type = standard then
14             data ← performRequest(order)
15             store(data, order.expirationTime)
16         else
17             cooperativeOrders.push(order)
18         end
19     end
20 upon the arrival of a request r from client:
21     increment totalClientRequests
22     data ← getLocalCache(r)
23     if data is not null increment cachedRequests and return data to
        the client
24     for order in cooperativeOrders do
25         if match(order, r) then
26             data ← getFromCacheNode(order.cacheNode, r)
27             if data is not null increment cachedRequests and return
                data to the client
28         end
29     end
30     for cacheNode in cacheNodes do
31         data ← getFromCacheNode(cacheNode, r)
32         if data is not null increment cachedRequests, return data to
            the client and break
33     end
34     data ← performRequest(r)
35     if reactiveCaching then
36         store(data, presetTime)
37     end
38     return data to client
```

5. CACHE-IT implementation

This Section describes CACHE-IT implementation. We utilize industry-adopted applications to fulfill some framework components while the others perform specific CACHE-IT tasks, which we implemented ourselves.

A first version of an **Interface Translator** was proposed in [9] and its implementation, namely C3PO (**C**onverter of **OP**en A**PI** **SP**ecification to WoT **O**bjects), was detailed in [41]. It can convert RESTful Web services APIs documented through the OpenAPI Specification (OAS) [42] into WTs. The current implementation requires a formal description of the provider interface (including its endpoints, inputs, outputs, and parameters) for the translation process. The OAS provides a language-independent standard to describe RESTful interfaces using a JSON-based description, and it is the *de facto* standard for API documentation.

We adopted the ELK stack[1] to fulfill the role of the **History Transfer** and **the History Storage**, as it satisfies the requirements listed. The ELK stack is a set of open-source tools that provide a flexible and scalable platform for collecting and storing distributed `log` data. ELK is an acronym built with the union of its three main components — i.e., Elasticsearch, Logstash, and Kibana. Elasticsearch is employed for indexing and storing data; Logstash is a data processing pipeline that collects and parses the `log` data to be stored in Elasticsearch. Logstash and Elasticsearch together fulfill the role of History Storage. The History Transfer is implemented by Filebeat, a lightweight shipper that is used to collect and transfer log data to Logstash. It can be instantiated in practically any computational environment and utilizes a back-pressure sensitive protocol to send data to Logstash, thus preventing overloading.

While the ELK stack was chosen for its scalability and flexibility, many other alternatives follow different approaches, like Fluentd[2] and Graylog.[3] Fluentd is built around an extensible architecture. Like Logstash, it offers a unified logging layer, but where it distinguishes itself is in its pluggable architecture. Fluentd supports numerous input and output sources via plugins, enabling it to integrate with various systems without core modifications. Also, it uses a lightweight core and is written in Ruby and C, which might provide efficiency benefits in specific environments. Graylog is centered on simplicity and ease of use. While Elasticsearch can be used as a backend for both ELK and Graylog, Graylog provides a more streamlined setup process and a centralized management interface for various logging pipelines. A notable feature of Graylog is its built-in alerting and reporting capabilities, allowing users to generate insights from their log data more seamlessly. However, based on the particular requirements and design principles of CACHE-IT, the ELK stack was determined to be most aligned with our goals.

[1] https://www.elastic.co

[2] https://www.fluentd.org

[3] https://graylog.org

The Cache Manager invokes the caching strategy functions through a POST request towards a user-specified address (defined in the Caching Template). The caching strategy needs to be wrapped in an application that exposes a REST API. That way, we preserve CACHE-IT separation of concerns and provide the user a rapid way to deploy and test different versions of their caching strategy since REST API can be hosted through a lightweight virtualization technique. This design allows caching strategy functions to implement more complex operations, including components tailored to specific scenarios or domains, such as mobility prediction. Finally, **Cache Worker** and the **Cache Manager** were developed as NodeJS[4] applications to be executed in a container environment [43].

As for the cache storage, we adopted Redis,[5] considered the *de facto* standard database for caching. It is a lightweight key–value store implementation that operates entirely in memory, making it suitable for use cases demanding low latency. Additionally, Redis currently supports various operating systems and hardware architectures, and its low resource requirements make it compatible with edge scenarios. The framework is designed to be deployed using lightweight virtualization (e.g., Docker containers[6]).

6. Validation

This section evaluates different CACHE-IT features under different client behaviors. Specifically, we aim to quantify the effects of (1) the number of neighbors visited by the cNN ranging from zero to two, (2) the caching strategy accuracy, and (3) the usage of cooperative caching orders as opposed to standalone orders. The objective is to understand how these factors impact the overall system performance, demonstrating the flexibility, applicability, and performance of CACHE-IT.

We designed and implemented an open-source simulator [44] to perform large-scale simulations by modeling the Cache Workers, the Cache Manager, and the network behavior.

In the experiments, we do not implement an explicit caching strategy. Instead, we emulate the caching strategy accuracy as a percentage that dictates the number of requests it could predict. Introducing a specific caching strategy would divert the evaluation toward the effectiveness of that particular strategy, overshadowing the evaluation of the cache architecture as a whole. By abstracting the specific caching strategy operations from our evaluation, we isolate the impact of the unique features and benefits provided by CACHE-IT. To emulate the caching strategy, the Cache Manager probabilistically selects a number of requests according to the caching strategy accuracy, sets them as a simulation input, and transforms those into standard caching orders. The caching order execution time is determined by the request execution time subtracted from a Gaussian time to represent the lack of precision in determining the specific request arrival time. The request expiration time is set to a default value of 10 min. If the experiment utilizes cooperative orders, the Cache Manager analyzes the caching orders and identifies duplicate resources cached in the same time slot; in those cases, it converts one of the orders to cooperative, pointing to the Cache Node that holds the resource in that time. Finally, the Cache Manager performs a redundancy check to eliminate duplicate caching orders. Standalone caching orders that store resources used by cooperative orders are excluded from this process.

For simplicity and without loss of generality, the data providers are abstracted as entities that generate resources over time. Its application and network behavior are modeled based on real datasets. We utilized [45] to model the network latency between the edge nodes and the data providers. The mentioned work characterizes cloud-to-user latency from different geographically distributed vantage points towards several data centers in distinct locations on the infrastructures of Amazon Web Services[7] and Microsoft Azure[8]. Upon initialization, each data provider randomly selects a specific cloud data center and emulates its network latency. We utilized the encrypted web traffic dataset as a base to model the data providers' behavior [46], specifically, the returned data size in bytes and the application processing time. The dataset comprises 800 real web services monitoring data — including the mentioned metrics. We select three significant web applications to represent high, medium, and low values regarding the returned data size of the data providers since this feature had non-overlapping distributions as opposed to processing time, which is similar to all services. The characterization of the selected web services is shown in Fig. 3 and in Fig. 4 regarding processing time and bytes returned. Data providers were assigned to each category equally. We disregard the latency added by the Device Abstraction Layer since a previous work [9] demonstrates it is insignificant.

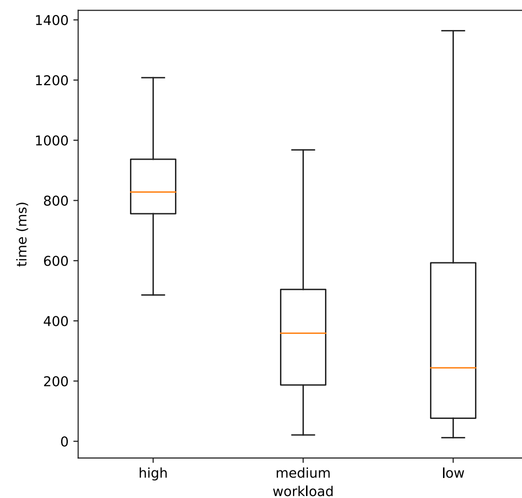


Fig. 3. Data characterization of processing time for the three categories of data providers.

The simulator deploys a configurable number of edge nodes within the experiment dimensions in random positions. These edge nodes serve as network gateways — e.g., base stations. Each edge node is equipped with a dedicated cache storage, and the caching replacement strategy employed in our experiments is the Least Recently Used (LRU) algorithm. We modeled the network between the client and the edge node following the wireless latency dataset present in [47], and the inter-edge nodes communications were modeled using the wired LAN dataset of the same source [47].

Clients in IoT scenarios often are mobile, ranging from industry 4.0 devices [17] to intelligent vehicles [20], so we add client mobility in the evaluation scenario. Each client moves in a trajectory determined by a list of random reference points within the experiment dimensions. The simulator generates a list of requests for each client for the experiment duration, and clients perform these requests by querying its closest edge node. The inter-arrival times between requests follow an exponential distribution corresponding to a Poisson process. The choice of which provider to query follows a Zipf popularity distribution since the latter is widely used to simulate the popularity of requests to data providers in the context of edge caching [6,12,20,21,27,30,32,48]. We utilized the Zipf parameter as 1.1 [32] independent of the client category. Additionally, we modeled three different patterns of client

[4] https://nodejs.org
[5] https://redis.io
[6] https://www.docker.com/
[7] https://aws.amazon.com/
[8] https://azure.microsoft.com/

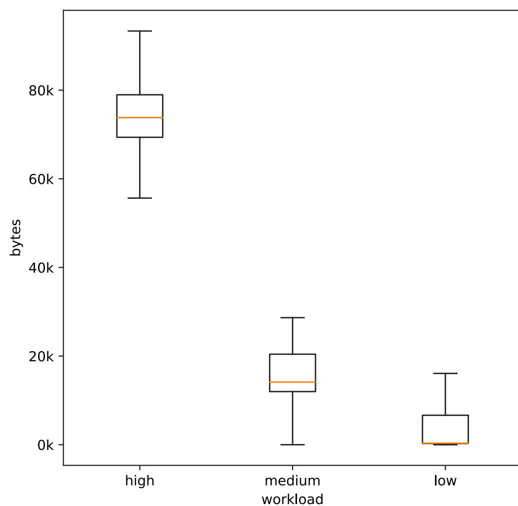


Fig. 4. Data characterization of response size in bytes for the three categories of data providers.

behavior based on observation in IoT systems; each client category alters how providers are selected. The popularity distribution in this context means that the popularity order of each data provider was randomly shuffled while still adhering to the same underlying Zipf distribution. The categories are:

- **ID**: clients in this category are independent of each other and follow their particular popularity distribution.
- **type**: Each client in this category is assigned a specific type with its own popularity distribution. This behavior was modeled assuming that devices of the same type (e.g., a specific brand and model of UAV) tend to consume data from similar providers.
- **location**: clients in this category have popularity distributions associated with their respective areas. The simulation considers the total area and divides it into a parameterized number of subareas, each with its popularity distribution.

Each category is illustrated in Fig. 5. We perform experiments for each client type and an experiment configuration in which we deploy all client types in the same experiment equally distributed — this experiment configuration is called "mix".

During the simulation, we track and record the following metrics:

- **Latency**: captures the time taken for each request to be returned to the client.
- **AoI**: represents the time difference between the generation of a resource and its arrival to the clients.
- **Number of Requests to Providers**: the total number of requests sent to the data providers.
- **Cache Hit Rate**: the percentage of requests retrieved from the cache as opposed to those forwarded to the data provider.

Finally, Table 4 lists the experiment parameters kept constant in all simulations. Table 5 depicts the factors and levels utilized in the performance evaluation. We perform experiments combining all the levels and factors. The caching order type denotes experiments executed only with standalone orders instead of the ones in which cooperative orders were used. For clarity, we refer to the experiments utilizing standard and cooperative orders as "cooperative". Each experiment was replicated 30 times, and the calculated confidence interval was 99%. In each replication, all entities involved were re-instantiated, increasing the inter-experiment variability — e.g., the clients' trajectories and initial positions, the edge nodes placement, the assignment of providers to their category, and a network trace.

Table 4
Experiment parameters.

Property	Value
Experiment Duration	1 h
Area dimensions	10 000 units^2
Number of edge nodes	10
Edge storage size	4 GB
Minimum distance between edge nodes	1000 units
Number of clients	500
Client Speed	10 units/s
Number of client types	5
Number of subareas	5
Default resource expiration time	10 min
Reactive caching	✓
Number of providers	750
Rate of requests per client (Poisson λ)	0.1 event/s
Popularity distribution (Zipf α)	1.1

Table 5
Factors and levels.

Factor	Level
Caching Strategy Accuracy	0%, 20%, 40%
Caching Orders Type	standalone, cooperative
c-NN (N neighbors visited)	0, 1, 2
Client Category	location, ID, type, mix

Fig. 6 depicts a handful selection of the most significant experiment configurations, which allows for a comprehensive analysis of the performance and behavior of our proposed framework. Those results were all executed with "mix" as the client category. We adopt a consistent naming pattern for our experiments: "accX-NY-Z", where X represents the accuracy percentage, Y denotes the number of neighbors visited (for the c-NN strategy), and Z indicates the caching order employed, "s" for standalone and "c" for cooperative. Exceptions are "baseline", which denotes the experiments without cache, and "regular-cache", which refers to experiments only applying the current reactive caching. The results showcase that the greater the caching strategy accuracy and the number of neighbors visited, the better the outcome for all metrics, except for AoI — since cached resources are less fresh than those fetched directly from the data providers.

It is noteworthy that CACHE-IT impacts positively in terms of latency, hit rate, and the number of requests sent to providers. The framework features enhance caching strategy accuracy, and we numerically show that the c-NN mechanism meaningfully improves the system. One experiment parameter that influences the results is the default time to keep resources cached (10 min), as the clients tend to query the same provider, which increases the overall hit rate, though it also increases the AoI. The baseline configuration value in Fig. 6, approximately 0.25 s, reflects the minimal time difference from resource generation to client delivery, as the resource was not cached for a period of time. System administrators deploying CACHE-IT must consider the trade-off between lower latency and data freshness. In scenarios where it is key to have low AoI, the amount of time to keep resources cached should be bounded by the maximum AoI.

Fig. 7 deepens the comparison between the experiments that use cooperative orders and experiments that do not. In general, the outcome of those experiments differs for the number of requests sent to providers and AoI. The number of requests is lower due to the cooperative aspect of the system sharing the predicted resource without performing additional requests. Consequently, the resources returned are less fresh, which increases the AoI. The difference in latency and hit rate between cooperative and standalone caching is attributed to removing redundant caching orders. Removing redundant caching orders does not eliminate caching orders that serve as pointers for cooperative ones, resulting in a higher number of requests and a greater average number of cached resources per Cache Node. However, when the number of visited neighbors (N) increases, this aspect is mitigated as resources are shared with the closest nodes.

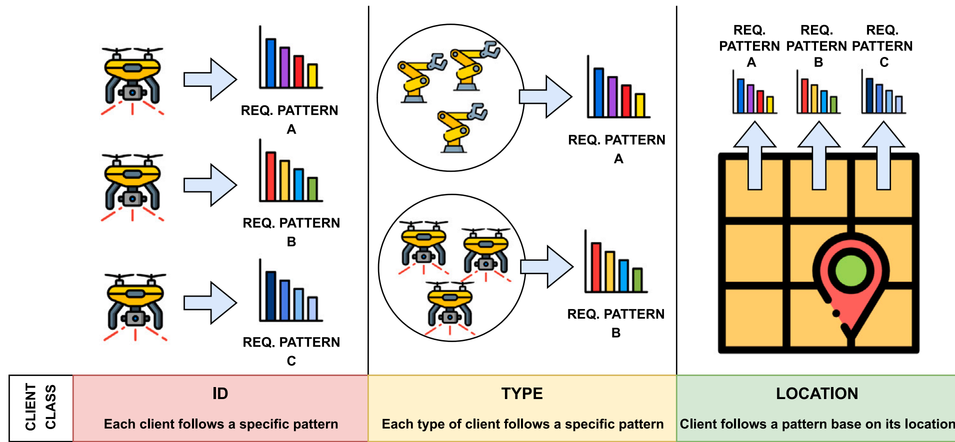


Fig. 5. Representation of the different categories of client behavior modeled in the experiments.

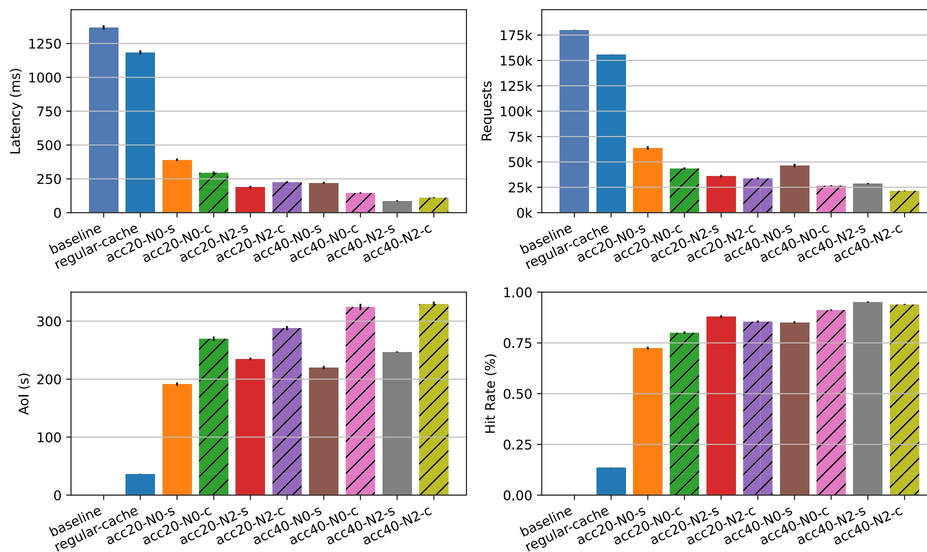


Fig. 6. Overall Simulation results for CACHE-IT comparing different configurations. Hatched bars represent experiments in which cooperative caching orders were used.

That behavior is observed by comparing the different Fig. 7 rows — each row corresponds to a number of neighborhoods visited, ranging from zero to two. This disparity in latency arises due to the additional one-hop requirement for cooperative orders to retrieve the requested data. When comparing the experiments with cNN equals two, both cooperative and standalone caching strategies exhibit similar high hit rates. However, the incremental latency introduced by cooperative orders (i.e., the additional hop to retrieve a resource) becomes more noticeable. In our experiments with ten edge nodes, configurations with two visited neighbors had access to approximately 30% of the total cached resources.

Fig. 8 shows sets of graphs characterizing the performance of the different client categories: each row represents a c-NN value, starting from zero until two. All the experiments were performed using standalone orders only. The results show that the client categories “type” and “location” have the best overall results since clients share a similar requesting pattern. In contrast, the “id” client category had slightly worse results since each client is independent of the other. Due to the heterogeneity of clients’ behaviors, the experiments with all the types – i.e., “mix” – follow the same pattern as observed in the experiments performed with the “id” client category, which means that in those scenarios, the clients’ popularity distribution was, in practice, independent from each other. However, Fig. 8 makes evident that the client category differences are minor when compared to the impact wielded by other simulation configurations, as the caching strategy accuracy. This behavior indicates that CACHE-IT is versatile enough to be exploited in different contexts regarding client interactions. The results presented in this section are open-source to guarantee transparency and replicability; they can be found in the project’s GitHub repository [44].

7. SHM use case

This section aims to validate the CACHE-IT architectural framework in an SHM case study, employing an IoT-based toolchain for real-time monitoring and diagnostic of a physical structure. We illustrate the customization potential of CACHE-IT by employing a specialized caching strategy that considers domain information and data source characteristics. This scenario is a comprehensive expansion of the case study initially demonstrated and deployed in our previous work [19].

Our exploration is divided into three subsections for easy comprehension and coherence. In Section 7.1, we offer a detailed overview of the scenario and clarify the caching goal we aim to accomplish. Next, in Section 7.2, we detail the Caching Template utilized, providing insights into the careful design choices and trade-offs we made to optimize performance. Finally, in Section 7.3, we present the outcomes of a small-scale performance analysis, in which we numerically demonstrate the effectiveness of the CACHE-IT framework in this context.

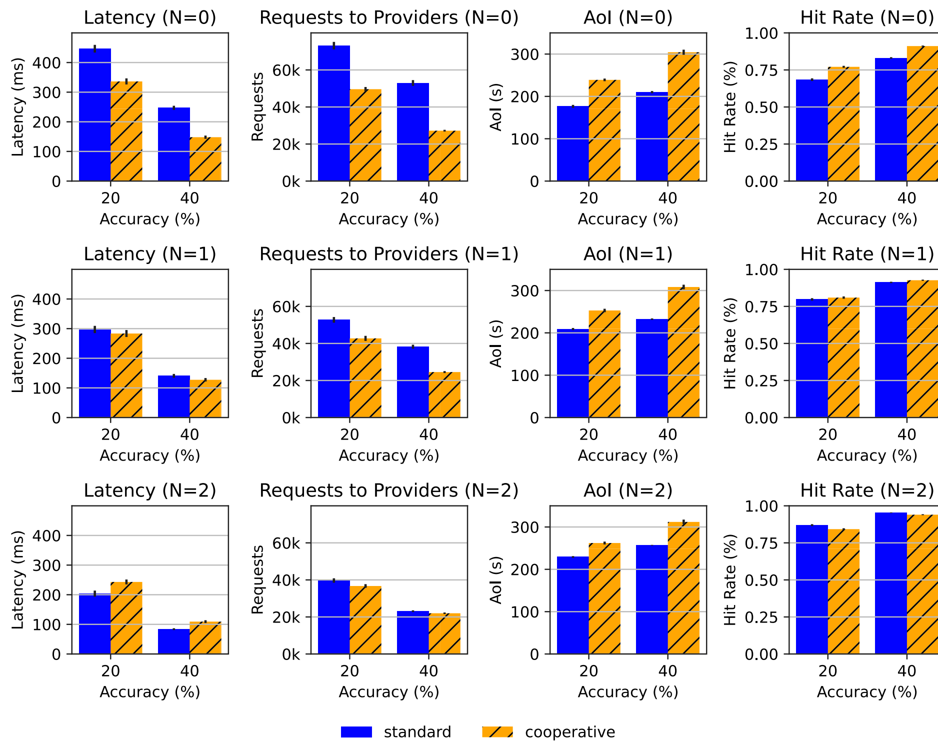


Fig. 7. Simulation results for CACHE-IT comparing standard and cooperative caching orders. Each row represents a different cNN configuration, denoted as N.

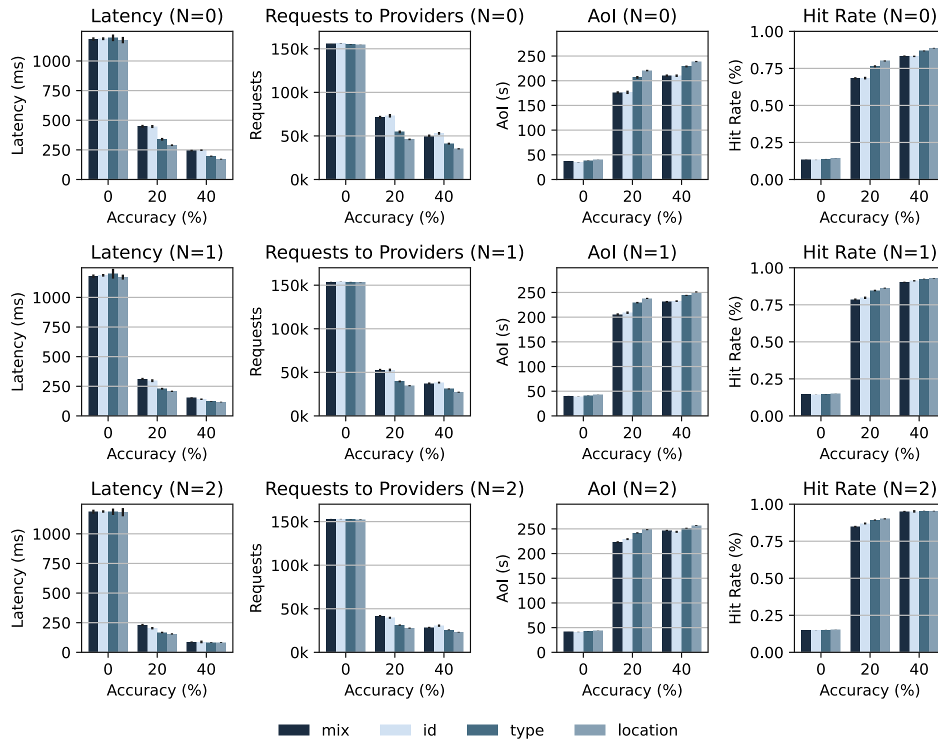


Fig. 8. Simulation results for CACHE-IT for different client types. Each row represents a different cNN caching configuration, denoted as N.

7.1. Caching goal and scenario description

The scenario revolves around an experimental 1:4 scale bridge with heterogeneous sensors forming a complex IoT network. The deployment, illustrated in Fig. 9, includes multiple peripheral sensor nodes, each hosting an accelerometer permanently attached to the structure and several cluster heads orchestrating subsets of the sensor network. These cluster heads, equipped with a wireless connection, battery, and a photovoltaic energy harvester, play a critical role in energy management, aiming to leverage the energy harvester to its maximum potential to prevent battery depletion. For this purpose, the harvester's external Digital Twin (DT) has been developed and deployed to estimate the devices' battery lifetime based on their electrical characteristics and current environmental conditions, specifically solar irradiance. Clients

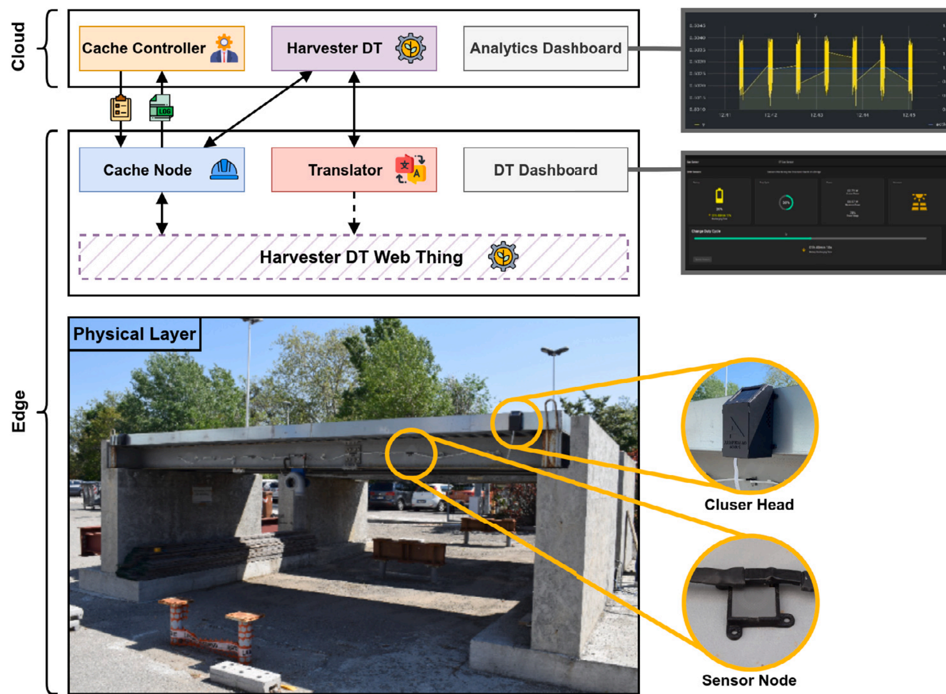


Fig. 9. The CACHE-IT framework deployed in a SHM case-study.

and applications can interact with the DT to optimize the device's duty cycle, extending battery life without disrupting the system's overall operation. However, the existing DT has limitations: it requires considerable processing power. It takes approximately 25 s on average to respond to a request due to low-level electronic interaction simulation for each request. This long processing time leads to waiting times that often extend to several hours, especially when several duty cycle simulations for each device are required. Adding new devices to the system further escalates this problem, imposing high latency that impacts the system in two ways: changes in environmental conditions before the DT reply and a poor user experience due to high screen loading times. To counteract these issues, a caching layer was designed and deployed to cache the outputs from the harvester DT.

The Harvester DT was deployed on a third-party server, over which we do not have direct control. The Cache Controller and the Analytics Dashboard, crucial components for managing and monitoring the cache system, were deployed on a cloud-like cluster composed of two machines. Each server is powered by a 16-core Intel CPU backed by 64 GB of RAM. On the other hand, the Cache Node, Translator, and DT Dashboard were deployed on an edge server, specifically a Raspberry Pi 3, located close to the physical structure. This setup demonstrates a C2T continuum deployment, with the `CACHE-IT` framework facilitating efficient communication between the components and ensuring optimal operation despite the inherent limitations of the devices, such as the Raspberry Pi's constrained resources.

7.2. Caching template definition

This testbed caching strategy relies on domain knowledge; the data requests to the DT contain information such as the device's electronic features, battery percentage, and the current solar irradiance, which is logged at the edge and transferred to the Cache Controller by the History Transfer. We designed the caching strategy to leverage those data to estimate each device's battery and solar irradiance for the next period and produce caching orders instructing those resources' caching. The other relevant Caching Template configurations are the functions that define the caching strategy:

- gen function: we choose the Prophet algorithm as the underlying technique since it is capable of forecasting time series data [49]. The gen function triggers the `fit` process, which outputs a model capable of generating predictions.
  The function extracts the historical solar irradiance and the battery level data points from the `log` and fits them into the Prophet model. This fitting process allows Prophet to understand the trends and seasonality present in our data. Once fitted, the model can generate a forecast for a future period. This forecast becomes the function $cOrders$ that we utilize to generate caching orders.
- $trigger$ function: this function is implemented to simply return a *true* value every 12 h. The periodic reevaluation is chosen to ensure that our caching strategy could respond to changes in demand patterns that might occur from day to day but without causing unnecessary computational overhead by recomputing the strategy too frequently.
- $period$ function: this function is designed to output the values from the last 7 days. This choice was based on the results obtained in [19]. This time frame balances the need for a sufficiently large dataset to capture trends and patterns with the need to keep the computational demands of the strategy manageable.

7.3. Results

We executed a comprehensive evaluation in the real testbed to quantify the impact of the `CACHE-IT` in the proposed environment. In our evaluation, we compared the performance of three distinct configurations:

- `no-cache`: This is the most basic configuration that acts as our baseline scenario. It does not implement any form of caching mechanism.
- `simple-cache`: This configuration emulates a traditional caching system where the responses from the Harvester DT are cached on the edge device.
- `CACHE-IT`: This is our proposed advanced caching strategy that leverages specific functions outlined in Section 7.2 of the study.

Table 6
CACHE-IT Results for the SHM use case.

Strategy	Latency (s)		Hit rate (%)	
	Avg	CI	Avg	CI
no-cache	29.72	[28.29, 31.15]	0	–
simple-cache	18.74	[16.85, 20.63]	41.70	[37.55, 45.85]
CACHE-IT	0.19	(0, 0.42]	98.91	[97.8, 100]

The evaluation generated a syntactic workload regarding the operation of ten cluster heads for 12 h and was replicated twenty times to ensure consistency in the findings. The edge caching, conversely, was deployed on a computational node possessing limited processing power — i.e., the mentioned Raspberry Pi. To emulate realistic network conditions, we generated requests synthetically following a Poisson distribution from a personal computer. The traffic generator and edge caching devices were collocated within the same LAN and communicated via WiFi. The traffic generator was in charge of emulating the cluster head battery depletion. At the same time, the irradiance data was modeled utilizing the National Solar Radiation Database [50] considering the location of the metropolitan area of Bologna. We selected random start dates and times from the solar irradiance database for each experiment replication to ensure a varied and unbiased representation.

Table 6 summarizes the results. Using the no-cache configuration, we experienced an average latency of 29.72 s due to the constant querying of the Harvester DT from the edge. As expected, since no caching was involved, the hit rate was 0%. The simple-cache notably reduced the average latency to 18.74 s, significantly improving from the no-cachescenario. The hit rate was 41.70%, meaning that almost half of the requests could be served directly from the cache, bypassing the need for a full round trip to the Harvester DT. The high hit rate of the simple-cache solution makes it evident that the scenario is composed of similar requests. Finally, our proposed CACHE-IT strategy outperformed both the previous strategies by a significant margin. The average latency drastically dropped to 0.19 s, and the hit rate increased to 98.91%, demonstrating that nearly all the requests were accurately predicted and cached on the edge. This case study depicts CACHE-IT potential to address custom caching strategies that rely on specific domain and application knowledge.

8. Conclusions and future work

In this work, we have presented CACHE-IT, a flexible and customizable framework for edge caching that addresses the features and challenges particular to IoT environments. Our framework decouples the caching strategy algorithm from the underlying architecture, enabling easy modification of caching strategies based on application-specific requirements. Further, it considers the C2T continuum for efficient resource allocation, exploring the computational power from the cloud while benefiting from the edge advantages such as low latency. Finally, we present novel mechanisms to handle dynamism by automatically triggering the update of the caching strategy, and we incorporate cooperation features in the framework that allow edge nodes to share resources and, thus, increase performance. Through extensive simulations, we demonstrated the effectiveness and performance of CACHE-IT in optimizing latency, improving cache hit rates, and reducing the number of requests to data provided . We deployed CACHE-IT in a real IoT testbed, which illustrated the customization of the framework by supporting a scenario-oriented caching strategy that considered use case specific information, which decreased the average system latency by more than 95%. In future works, we plan to develop caching strategies based on the prevalent techniques in the literature to perform proactive edge caching (e.g., deep learning) and strategies that aim to optimize AoI. Our objective is to establish an open marketplace where users can select and customize caching strategies as off-the-shelf resources. We believe that this approach has the potential to foster an engaging community that actively contributes to the enhancement of the CACHE-IT framework. Furthermore, we plan to integrate federated learning into the framework's architecture. This integration would address potential privacy concerns associated with centralized data collection.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Marco Di Felice reports financial support was provided by National Institute for Insurance against Accidents at Work.

Data availability

Data will be made available on request.

Acknowledgments

The presented work has been partially supported by the project "DS2: Digital Smart Structures", funded by INAIL, Italy (Italian Workers' Compensation Authority), BRIC 2021. The icons in the images are courtesy of FlatIcon.

References

[1] P. Sharma, S. Jain, S. Gupta, V. Chamola, Role of machine learning and deep learning in securing 5G-driven industrial IoT applications, Ad Hoc Netw. 123 (2021) 102685, http://dx.doi.org/10.1016/j.adhoc.2021.102685.
[2] F. Zonzini, C. Aguzzi, L. Gigli, L. Sciullo, N. Testoni, L. De Marchi, M. Di Felice, T.S. Cinotti, C. Mennuti, A. Marzani, Structural health monitoring and prognostic of industrial plants and civil structures: A sensor to cloud architecture, IEEE Instrum. Meas. Mag. 23 (9) (2020) 21–27.
[3] M. Mohammadi, A. Al-Fuqaha, S. Sorour, M. Guizani, Deep learning for IoT big data and streaming analytics: A survey, IEEE Commun. Surv. Tutor. 20 (4) (2018) 2923–2960, http://dx.doi.org/10.1109/COMST.2018.2844341.
[4] X. Li, J. Wan, Proactive caching for edge computing-enabled industrial mobile wireless networks, Future Gener. Comput. Syst. 89 (2018) 89–97, http://dx.doi.org/10.1016/j.future.2018.06.017.
[5] F. Righetti, C. Vallati, S.K. Das, G. Anastasi, An experimental evaluation of the 6top protocol for industrial IoT applications, in: 2019 IEEE Symposium on Computers and Communications, ISCC, 2019, pp. 1–6, http://dx.doi.org/10.1109/ISCC47284.2019.8969590.
[6] R.W.L. Coutinho, A. Boukerche, Modeling and analysis of a shared edge caching system for connected cars and industrial IoT-based applications, IEEE Trans. Ind. Inform. 16 (3) (2020) 2003–2012, http://dx.doi.org/10.1109/TII.2019.2938529.
[7] M. Chiang, S. Ha, F. Risso, T. Zhang, I. Chih-Lin, Clarifying fog computing and networking: 10 questions and answers, IEEE Commun. Mag. 55 (4) (2017) 18–20, http://dx.doi.org/10.1109/MCOM.2017.7901470.
[8] K. Cao, Y. Liu, G. Meng, Q. Sun, An overview on edge computing research, IEEE Access 8 (2020) 85714–85728, http://dx.doi.org/10.1109/ACCESS.2020.2991734.
[9] I. Zyrianoff, L. Gigli, F. Montori, C. Aguzzi, S. Kaebisch, M. Di Felice, Seamless integration of restful web services with the web of things, in: 2022 IEEE International Conference on Pervasive Computing and Communications Workshops and Other Affiliated Events, (PerCom Workshops), 2022, pp. 427–432, http://dx.doi.org/10.1109/PerComWorkshops53856.2022.9767531.
[10] J. Yao, T. Han, N. Ansari, On mobile edge caching, IEEE Commun. Surv. Tutor. 21 (3) (2019) 2525–2553, http://dx.doi.org/10.1109/COMST.2019.2908280.
[11] L. Ale, N. Zhang, H. Wu, D. Chen, T. Han, Online proactive caching in mobile edge computing using bidirectional deep recurrent neural network, IEEE Internet Things J. 6 (3) (2019) 5520–5530, http://dx.doi.org/10.1109/JIOT.2019.2903245.
[12] S. Rathore, J.H. Ryu, P.K. Sharma, J.H. Park, DeepCachNet: A proactive caching framework based on deep learning in cellular networks, IEEE Netw. 33 (3) (2019) 130–138, http://dx.doi.org/10.1109/MNET.2019.1800058.
[13] T.-V. Nguyen, N.-N. Dao, V. Dat Tuong, W. Noh, S. Cho, User-aware and flexible proactive caching using LSTM and ensemble learning in IoT-MEC networks, IEEE Internet Things J. 9 (5) (2022) 3251–3269, http://dx.doi.org/10.1109/JIOT.2021.3097768.
[14] M. Pappalardo, E. Mingozzi, A. Virdis, A model-driven approach to aol-based cache management in IoT, in: 2021 IEEE 26th International Workshop on Computer Aided Modeling and Design of Communication Links and Networks, CAMAD, 2021, pp. 1–6, http://dx.doi.org/10.1109/CAMAD52502.2021.9617772.

[15] R. Singh, R. Sukapuram, S. Chakraborty, A survey of mobility-aware multi-access edge computing: Challenges, use cases and future directions, Ad Hoc Netw. 140 (2023) 103044, http://dx.doi.org/10.1016/j.adhoc.2022.103044.

[16] M. Noura, M. Atiquzzaman, M. Gaedke, Interoperability in internet of things: Taxonomies and open challenges, Mobile Netw. Appl. 24 (2019) 796–809.

[17] M. Pettorali, F. Righetti, C. Vallati, S.K. Das, G. Anastasi, Mobility management in industrial IoT environments, in: 2022 IEEE 23rd International Symposium on a World of Wireless, Mobile and Multimedia Networks, (WoWMoM), 2022, pp. 271–280, http://dx.doi.org/10.1109/WoWMoM54355.2022.00046.

[18] S. Zhang, H. Luo, J. Li, W. Shi, X. Shen, Hierarchical soft slicing to meet multi-dimensional QoS demand in cache-enabled vehicular networks, IEEE Trans. Wireless Commun. 19 (3) (2020) 2150–2162, http://dx.doi.org/10.1109/TWC.2019.2962798.

[19] I. Zyrianoff, A. Trotta, L. Sciullo, F. Montori, M. Di Felice, IoT edge caching: Taxonomy, use cases and perspectives, IEEE Internet Things Mag. 5 (3) (2022) 12–18, http://dx.doi.org/10.1109/IOTM.001.2200112.

[20] G. Yu, Y. He, J. Wu, Z. Chen, J. Pan, Mobility-aware proactive edge caching for large files in the internet of vehicles, IEEE Internet Things J. (2023) 1, http://dx.doi.org/10.1109/JIOT.2023.3240423.

[21] A.A. Chowdhury, I. Islam, M.I.A. Zahed, I. Ahmad, An optimal strategy for UAV-assisted video caching and transcoding, Ad Hoc Netw. 144 (2023) 103155, http://dx.doi.org/10.1016/j.adhoc.2023.103155.

[22] K. Thar, T.Z. Oo, Y.K. Tun, D.H. Kim, K.T. Kim, C.S. Hong, A deep learning model generation framework for virtualized multi-access edge cache management, IEEE Access 7 (2019) 62734–62749, http://dx.doi.org/10.1109/ACCESS.2019.2916080.

[23] D.T. Hoang, D. Niyato, D.N. Nguyen, E. Dutkiewicz, P. Wang, Z. Han, A dynamic edge caching framework for mobile 5G networks, IEEE Wirel. Commun. 25 (5) (2018) 95–103, http://dx.doi.org/10.1109/MWC.2018.1700360.

[24] M.U. Farooq, M. Zeeshan, M.T. Jahangir, M. Asif, A novel cooperative micro-caching algorithm based on fuzzy inference through NFV in ultra-dense IoT networks, J. Netw. Syst. Manage. 30 (1) (2021) 20, http://dx.doi.org/10.1007/s10922-021-09632-6.

[25] F. Zhang, G. Han, L. Liu, M. Martínez-García, Y. Peng, Joint optimization of cooperative edge caching and radio resource allocation in 5G-enabled massive IoT networks, IEEE Internet Things J. 8 (18) (2021) 14156–14170, http://dx.doi.org/10.1109/JIOT.2021.3068427.

[26] S. Xu, X. Liu, S. Guo, X. Qiu, L. Meng, MECC: A mobile edge collaborative caching framework empowered by deep reinforcement learning, IEEE Netw. 35 (4) (2021) 176–183, http://dx.doi.org/10.1109/MNET.011.2000663.

[27] X. Li, X. Wang, C. Zhu, W. Cai, V.C.M. Leung, Caching-as-a-service: Virtual caching framework in the cloud-based mobile networks, in: 2015 IEEE Conference on Computer Communications Workshops, (INFOCOM WKSHPS), 2015, pp. 372–377, http://dx.doi.org/10.1109/INFCOMW.2015.7179413.

[28] Y. Hao, Y. Miao, L. Hu, M.S. Hossain, G. Muhammad, S.U. Amin, Smart-edge-cocaco: AI-enabled smart edge with joint computation, caching, and communication in heterogeneous IoT, IEEE Netw. 33 (2) (2019) 58–64, http://dx.doi.org/10.1109/MNET.2019.1800235.

[29] C.K. Kim, T. Kim, S. Lee, S. Lee, A. Cho, M.-S. Kim, Delay-aware distributed program caching for IoT-edge networks, Plos One 17 (7) (2022) e0270183.

[30] X. Zhao, Q. Zhu, Mobility-aware and interest-predicted caching strategy based on IoT data freshness in D2D networks, IEEE Internet Things J. 8 (7) (2021) 6024–6038, http://dx.doi.org/10.1109/JIOT.2020.3033552.

[31] Y.M. Saputra, D.T. Hoang, D.N. Nguyen, E. Dutkiewicz, D. Niyato, D.I. Kim, Distributed deep learning at the edge: A novel proactive and cooperative caching framework for mobile edge networks, IEEE Wirel. Commun. Lett. 8 (4) (2019) 1220–1223, http://dx.doi.org/10.1109/LWC.2019.2912365.

[32] Y. Zhang, B. Feng, W. Quan, A. Tian, K. Sood, Y. Lin, H. Zhang, Cooperative edge caching: A multi-agent deep learning based approach, IEEE Access 8 (2020) 133212–133224, http://dx.doi.org/10.1109/ACCESS.2020.3010329.

[33] T. Li, L. Song, Federated online learning aided multi-objective proactive caching in heterogeneous edge networks, IEEE Trans. Cogn. Commun. Netw. (2023) 1, http://dx.doi.org/10.1109/TCCN.2023.3262243.

[34] Y. Zhang, Y. Li, R. Wang, J. Lu, X. Ma, M. Qiu, PSAC: Proactive sequence-aware content caching via deep learning at the network edge, IEEE Trans. Netw. Sci. Eng. 7 (4) (2020) 2145–2154, http://dx.doi.org/10.1109/TNSE.2020.2990963.

[35] D. Li, H. Zhang, H. Ding, T. Li, D. Liang, D. Yuan, User preference learning-based proactive edge caching for D2D-assisted wireless networks, IEEE Internet Things J. (2023) 1, http://dx.doi.org/10.1109/JIOT.2023.3244621.

[36] H. Wu, Y. Xu, J. Li, PTF: Popularity-topology-freshness-based caching strategy for ICN-IoT networks, Comput. Commun. 204 (2023) 147–157.

[37] H. Wu, J. Li, J. Zhi, Could end system caching and cooperation replace in-network caching in CCN? in: Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication, 2015, pp. 101–102.

[38] W3C Working Group, Wot reference architecture (W3C recommendation 9 April 2020), 2023, http://www.w3.org/TR/wot-architecture. (Accessed on 1 June 2023).

[39] M. Reiss-Mirzaei, M. Ghobaei-Arani, L. Esmaeili, A review on the edge caching mechanisms in the mobile edge computing: A social-aware perspective, Internet Things 22 (2023) 100690, http://dx.doi.org/10.1016/j.iot.2023.100690.

[40] D. Silva, A. Heideker, I.D. Zyrianoff, J.H. Kleinschmidt, L. Roffia, J.-P. Soininen, C.A. Kamienski, A management architecture for IoT smart solutions: Design and implementation, J. Netw. Syst. Manage. 30 (2) (2022) 35, http://dx.doi.org/10.1007/s10922-022-09648-6.

[41] I. Zyrianoff, L. Gigli, F. Montori, C. Aguzzi, S. Kaebisch, M. Di Felice, Artifact: C3PO - converter of open API specification to WoT objects, in: 2022 IEEE International Conference on Pervasive Computing and Communications Workshops and Other Affiliated Events, (PerCom Workshops), 2022, pp. 185–186, http://dx.doi.org/10.1109/PerComWorkshops53856.2022.9767293.

[42] Swagger, Openapi specification 3.1.0, 2021, https://swagger.io/specification/. (Accessed on 1 June 2023).

[43] I. Zyrianoff, CACHE-IT cache worker, https://github.com/ivanzy/cache-worker.

[44] I. Zyrianoff, CACHE-IT simulator, https://github.com/UniBO-PRISMLab/cache_simulator.

[45] F. Palumbo, G. Aceto, A. Botta, D. Ciuonzo, V. Persico, A. Pescapé, Characterization and analysis of cloud-to-user latency: The case of azure and AWS, Comput. Netw. 184 (2021) 107693, http://dx.doi.org/10.1016/j.comnet.2020.107693.

[46] [dataset], S. Špaček, P. Velan, P. Čeleda, D. Tovarňák, Encrypted web traffic dataset: Event logs and packet traces, Data Brief 42 (2022) 108188, http://dx.doi.org/10.1016/j.dib.2022.108188.

[47] [dataset], G.A. Stafford, LAN network stability, 2023, https://www.kaggle.com/datasets/garystafford/ping-data. (Accessed on 31 May 2023).

[48] Z. Sang, S. Guo, Q. Wang, Y. Wang, GCS: Collaborative video cache management strategy in multi-access edge computing, Ad Hoc Netw. 117 (2021) 102516, http://dx.doi.org/10.1016/j.adhoc.2021.102516.

[49] S.J. Taylor, B. Letham, Forecasting at scale, Amer. Statist. 72 (1) (2018) 37–45.

[50] [dataset], A.M. Gracia Amillo, N. Taylor, M.A. M., E.D. Dunlop, P. Mavrogiorgios, F. Fahl, G. Arcaro, I. Pinedo, Adapting PVGIS to trends in climate, technology and user needs, in: 38th European Photovoltaic Solar Energy Conference and Exhibition, PVSEC, 2021, pp. 907–911.

Ivan Zyrianoff is a Ph.D. student from the University of Bologna and a member of the IoT-Prism lab. He holds a B.S. degree in Computer Science and an M.S. degree in Information Engineering from the Federal University of ABC, Brazil. He was involved in the SWAMP Project, an EU-Brazil collaborative research project that developed IoT-based methods and approaches for smart water management in precision irrigation. He also participated in the Arrowhead Tools project, which aims for the digitalization and automation solutions for the European industry. His current research topics encompass interoperability for the Internet of Things and Edge Computing.

Lorenzo Gigli received his Master's Degree (summa cum Laude) in Computer Science in 2019 from the University of Bologna, Italy. He worked as a Research Fellow on the MAC4PRO project (INAIL) at the Department of Computer Science and Engineering (DISI), University of Bologna, Italy. He is enrolled in a Ph.D. program in Engineering and Information Technology. He is part of the IoT PRISM laboratory directed by Prof. Marco Di Felice. His field of study includes the Internet of Things, Blockchains, and Distributed Systems.

Federico Montori received the B.S. and M.S. degrees (summa cum laude) in computer science and the Ph.D. degree in computer science and engineering from the University of Bologna in 2012, 2015, and 2019, respectively. He was a Visiting Researcher with the Swinburne University of Technology, Australia, and Luleå Tekniska Universitet, Sweden. He is currently an Assistant Professor with the University of Bologna. He participated in several EU projects and he was WP Leader for the H2020 Project Arrowhead Tools. His primary research interests include mobile crowdsensing (MCS), pervasive and mobile computing, the IoT automation, and IoT data analysis.

Luca Sciullo received the master's degree (summa cum laude) in computer science and the Ph.D. degree in computer science and engineering from the University of Bologna, Italy, in 2017 and 2021, respectively. He is a Junior Assistant Professor with the University of Bologna. He was a Visiting Researcher at the Huawei European Research Center of Munich, Germany. He is a part of the IoT Prism Laboratory directed by Prof. Marco Di Felice and Prof. Luciano Bononi. His research interests include wireless systems and protocols for emergency scenarios, wireless sensor networks, the IoT systems, and the Web of Things.

Carlos A. Kamienski is a Full Professor of Computer Science at the Federal University of ABC (UFABC, Brazil). For eight years, he led the NUVEM Strategic Research Unit comprising faculty members and students working in smart societies, virtual sensations, connected mobility, extreme computing, and integrated universes. He was the Brazilian coordinator of SWAMP from 2017 to 2021 (swamp-project.org), an EU-Brazil collaborative research project that developed IoT-based methods and approaches for smart water management in precision irrigation. His current research interests include the Internet of Things, smart agriculture, smart cities, fog computing, network softwarization, and Future Internet.

Marco Di Felice received his Laurea and Ph.D. degrees in computer science from the University of Bologna in 2004 and 2008, respectively. He has held visiting research positions at the Georgia Institute of Technology and Northeastern University. Currently, he is a Full Professor of computer science at the University of Bologna and serves as the Co-Director of the IoT PRISM Laboratory. With over 150 papers published, his research focuses on wireless and mobile systems, including self-organizing networks, unmanned aerial systems, IoT, WoT, and context-aware computing. He has been recognized with three Best Paper Awards and is an Associate Editor for the IEEE Internet of Things Journal.