CATER: A Policy-Based Data Placement Framework for Edge Storage

Ahmed Khalid, Sean Ahearne
Dell Research, *Dell Technologies*, Cork, Ireland
{ahmed.khalid, sean.ahearne}@dell.com

Hemant K. Mehta, Utz Roedig, Cormac J. Sreenan
School of CSIT, *University College Cork*, Ireland
{h.mehta, u.roedig, cjs}@cs.ucc.ie

***Abstract*—The growing heterogeneity and decentralization in the modern computing paradigm of edge-cloud continuum introduces new constraints on storage systems, such as storage type, associated processors, privacy, scarce resources, compliance, GDPR and geographical restrictions. While existing distributed data and object stores can ensure data availability and fault-tolerance, they are not flexible or dynamic enough to address these diverse set of constraints. In this paper, we introduce a modular policy-driven data placement framework, `CATER`, designed to seamlessly integrate with existing storage systems and overcome the aforementioned limitations. `CATER` formulates the data placement problem as an optimization model, incorporating data collocation and hardware constraints. We integrated a prototype of `CATER` with Apache Ozone and conducted experiments and simulations. Results show a 23% improvement in data placement while respecting 100% of the constraints.**

I. Introduction

As the proliferation of edge devices continues to accelerate, the need for an efficient and adaptive data storage and placement framework becomes increasingly vital. In contrast to cloud computing, edge computing exhibits unique attributes that give rise to distinct data placement challenges and necessitate innovative and adaptive solutions. These characteristics include: low latency, proximity to data source, resource constraints, dynamicity, heterogeneity, stricter privacy and compliance requirements.

Our research has emanated from a large-scale research project that created a platform for data processing and artificial intelligence using network-edge computing. A core element of such an edge platform is a fault-tolerant storage system, with an underlying distributed architecture and advanced replication techniques [1]. Data may be stored as user-friendly files, fixed-size blocks, or objects with additional metadata. An edge storage solution must be able to address these heterogeneous storage type requirements.

Furthermore, edge clusters store data of diverse applications on multiple nodes leading to numerous constraints going beyond throughput [2]. For example, due to data sovereignty and GDPR, restrictions on physical storage location may arise [3]. The desire to optimize performance may result in requiring nodes with specific compute architectures, e.g. GPU or FPGA. To improve safety against side channel attacks [4] and to apply contractual obligations in multi-tenant settings, data owners may impose restrictions on data collocation.

These additional constraints must be met without compromising existing mechanisms for robust storage through replication, and while respecting the provider-specific objectives of storage systems to maintain a high level of efficacy and performance or to minimize energy consumption.

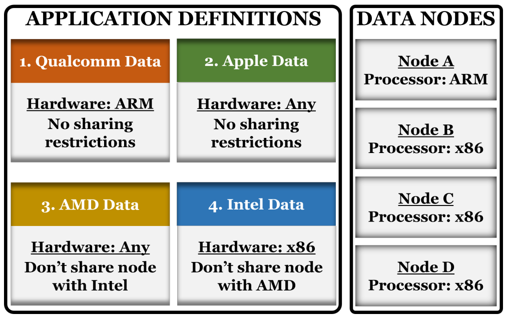


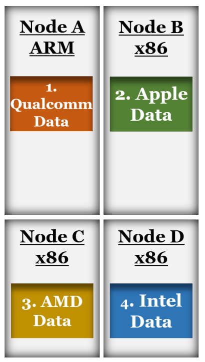
Fig. 1. Toy Example. A sample set of applications and datanode definitions.

To establish the need for a novel dynamic data placement framework in modern computing paradigms, we present a simplified example in a smart chip manufacturing factory floor [5] that produces chips for various competing vendors. Assembly lines are serviced by Edge Micro Datacenters (EMDC) [6] for processing and storage of data. These systems can be considered a small set of computers behaving as a distributed computing cluster, often comprising of 4-32 individual computing nodes. The applications of various vendors on these EMDC's may have constraints like reluctance to share disks containing their integrated circuit designs. A particular set of application requirements and state of datanodes will result in multiple possible data placement solutions. For simplicity, let's assume that each datanode would have sufficient space to store all applications together. Figure 1 lists various compute proximity and privacy requirements, e.g., Application 1 requires an ARM processor to process Qualcomm's data, hence it must be placed on Node A (the only node with ARM). Intel and AMD applications restrict their data to be collocated with each other.

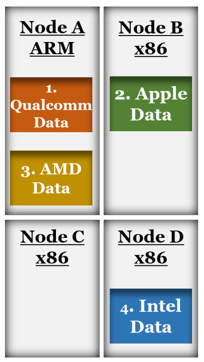
Assuming the objective of the storage provider is to maximize their energy savings by minimizing the number of active nodes, this example environment with all its constraints, leads to 27 feasible solutions in total. Three of these solutions are:

One application per node (Figure 2a): As there are four applications and four nodes, a naive solution is to put each application on a separate node in a way that doesn't violate any constraints. (E.g., $1 \rightarrow A$, $2 \rightarrow B$, $3 \rightarrow C$, $4 \rightarrow D$)

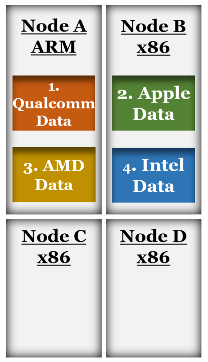
Move application 3 **to node** A (Figure 2b): As 3 can



(a) All applications on separate nodes.



(b) Energy saved as one node is inactive.



(c) All constraints met with two nodes.

Fig. 2. Three possible solutions for the simplified data placement problem.

run on any hardware and node A meets the rest of the requirements, moving 3 to A will free up node C, which can go to sleep mode and save energy, until a new application or data movement requires it to be reactivated.

Use only nodes A and B (Figure 2c): Placing Intel's data on node B provides the best solution that meets all criteria and uses only two nodes. Reducing active nodes further will violate collocation constraints.

Physically separating data provides security to a level beyond what software isolation mechanisms offer. Another example that reinforces the need for strict data collocation policies is an EMDC in a hospital, running applications and storing data from distinct parties, including patient medical records, hospital administration records, various medical insurance companies and device monitoring services.

These real-world scenarios provide a sense of the motivation for providing a solution for data placement framework and algorithms that can respect diverse constraints including data collocation and hardware proximity.

This paper presents `CATER`, a novel modular poliCy-bAsed daTa placEment fRamework, that offers a unique view and set of features to distinguish distributed data stores by allowing them to expose a range of sophisticated placement strategies in a dynamic heterogeneous environment. The specific research contributions of this paper are as follows:

- Design of a data placement framework, that can operate in an external and loosely-coupled manner with existing storage systems, and thus can be easily enhanced without changing the underlying system.
- An optimization model for the provider-defined objective of minimizing the number of active datanodes. Reducing active nodes leads to lower energy consumption for the edge servers [7]. The problem is formulated as an Integer Linear Program and solved using a Constrained Programming with CP-SAT solver [8].
- A heuristics algorithm for real-time data placement. This runs, after the initial optimal placement, to efficiently handle the modifications in the applications and constraints, and to reduce data movement between datanodes.
- A prototype implementation of `CATER` and integration with Apache Ozone [9]. This demonstrates the practical feasibility of our solution and allows us to evaluate the performance of the proposed placement algorithm.

The subsequent sections of the paper present the related work (Section II), reference architecture (Section III), problem definition and formulation (Section IV), implementation (Section V), and evaluation (Section VI).

II. Related Work

Data placement has generally been considered a sub-task of the workload placement problemm, which has been shown to have a significant impact on workload execution [10]. However, with the explosive increase in data volume and the advent of the Edge computing paradigm (where data takes precedence over compute), data placement has emerged as a separate scheduling task with its own distinct challenges. Specifically, the data placement problem differs from workload placement in the following ways:

- Data Locality: While mission-critical applications may suffer from network delays if workloads are located far away, e.g., in the cloud, the impact is much more significant for data. Hence, it is of key importance to minimize data transfer across the network by placing data close to the computing resources that need it, thus reducing latency, especially significant when the data volume is high [11].
- Consistency: Data is generally stored with multiple replicas. This requires ensuring that distributed copies of data remain consistent and coherent in the face of concurrent updates and access.
- Scalability: Handling data placement strategies that scale with an increasing volume of data and system nodes.
- Data Privacy: Managing data placement to comply with privacy and security regulations; ensuring sensitive data is stored appropriately [12].
- Heterogeneity: In addition to the same heterogeneity experienced by processing and compute, storage also faces significant other diverse constraints in terms of storage types, media, data types, storage formats, tiering, protocols, and storage models [13]. This makes the problem distinct enough to inspire data specific solutions and algorithms for data placement.

The rest of this section presents data storage and placement solutions that focus on specific objectives.

A. Data Storage Frameworks Targeting Performance

CoHadoop [14] proposed a technique to collocate the related data on the same set of datanodes. This technique is developed as an extension to Hadoop and focuses on a major performance bottleneck of Hadoop which is the ability to collocate related data on the same set of nodes. This technique requires input from the applications that some of the files are related and are supposed to be processed together.

To improve the job run-time, Tula [15] proposed a block placement technique and disk latency-aware balancer which ensures that low-latency disks get more blocks, in comparison to high-latency disks, for reduced overall disk latency, thus leading to improved job run times. Liu et al. proposed a group-based replica placement policy [16] for large-scale geospatial 3D raster data, to optimize the locations of the replicas, with the aim of reducing the network overhead and improving the cluster performance.

We note that these research contributions focused on improving performance of a data store, mostly Hadoop, rather than the problem of constraints-based data placement.

B. Data Storage Frameworks Targeting Sensitivity

HadoopSec [17] presents a *sensitivity*-aware data placement strategy for Hadoop using a machine learning-based approach for secure data placement. It was suggested that companies are concerned with building a single large cluster with data of multiple projects due to security vulnerabilities and privacy invasion by malicious attackers and internal users. HadoopSec also uses a rack-awareness script that utilizes the available information to find the affinity levels of nodes.

HadoopSec 2.0 is an extension of HadoopSec [18] that uses a prescriptive, adaptive, and intelligent system to identify patterns in the input data and group those with similar security concerns. Similarly, Sensitive Data Detection (SSD) is a framework proposed for identifying the sensitivity of data to be stored in Hadoop [19].

HadoopSec and HadoopSec 2.0 use data sensitivity-based constraints. In contrast, `CATER` deals with a wider variety of collocation constraints, and also datanode location, disk space on node, space demand of applications, and the node's architectural specification. This set of constraints can be easily extended, and, while currently `CATER` is implemented for Ozone, it is designed to be easily integrated with other storage systems, without being intimately tied to just one system.

III. Reference Placement Architecture

Our constraints-based data placement algorithm has been developed and implemented as an external component, following the modern application development style of Kubernetes and microservices. This loosely-coupled relationship between storage and placement component will make it easier to maintain and to update the placement algorithms, without changing the underlying storage system.

This component is part of the BRAINE system for enabling Edge AI [20], within which it interacts using APIs to retrieve application data constraints and policy information (forming the data management framework) and adapt the algorithm output to impact workload placement on edge nodes. Our data placement component interacts specifically with a front-end for specifying workloads and constraints, and a policy manager for storing constraints.

Figure 3 illustrates the workflow process, showing the individual steps involved. Users provide blueprints for the applications and associated requirements and data restrictions (Step 0). Blueprints are stored in a service registry and policy manager so that they can be modified and used for deployment when desired. Subsequently, a user requests to schedule an application (Step 1). If there are data placement constraints,

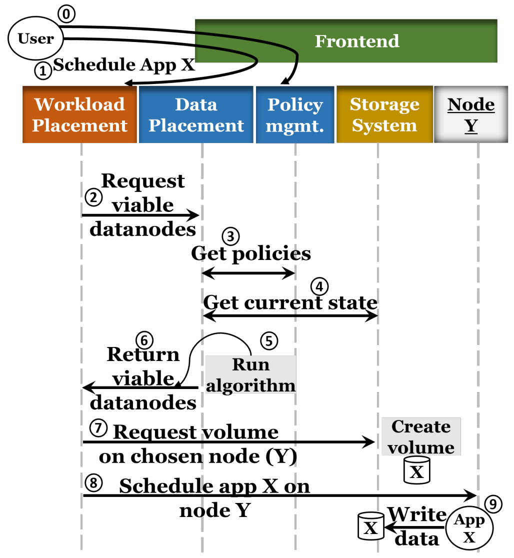


Fig. 3. Deployment Workflow.

step 2 initiates the data placement process, leading to the subsequent steps 3-9, as labelled in Figure 3.

Note that the workload and data volume may or may not be deployed on the same compute node, depending on the solutions found by the algorithm and the restrictions provided by the user or platform. In the event that the workload and data are not located on the same node this will not affect workload functionality as it is supported by the system architecture, but it may impact data access latency.

IV. Data Placement Problem and Solution

The modular nature of our data placement framework (Section III) makes it agnostic to the design of the placement algorithm and the objectives of a storage provider. For example, a provider may use the framework to run an algorithm that minimizes: (i) Latencies in storage access, (ii) Energy consumption, (iii) Storage cost, e.g., by allocating cheaper storage first, (iv) Bandwidth consumption, or the distance between data source and datanodes.

To demonstrate the efficacy of our placement framework, we formulate a novel optimization model that solves the data placement problem and minimizes the number of actively used nodes and consequently reduces energy consumption and carbon footprint. The model incorporates a diverse range of data collocation and hardware-related constraints. In the remainder of this section, we present our system model, formulate an optimization model to solve the problem, and design a heuristics algorithm for real-time computation.

A. System Model

We consider an ecosystem with a set of data storage nodes (denoted by D) each with a certain storage space available (denoted by T_j for node j). Datanodes are managed by a physically distributed but logically centralized storage system that has a global view on the system. A set of applications (denoted by A) require a certain amount of storage volume

TABLE I
NOTATION USED IN THE OPTIMIZATION MODEL.

Symbol	Description
INPUTS	
A	Set of applications that require storage volumes
D	Set of datanodes
R_i	Number of replicas required for application i
T_j	Total storage space available on datanode j
V_i	Volume size needed for application i
U_{xy}	Binary variable for collocation restrictions. 1 means x and y cannot be collocated
C_{ij}	Binary representing if application i is compatible with node j
OUTPUTS	
P_j	Binary variable to determine whether datanode j is powered-on (has an assigned application)
B_{ij}	Binary value determining if application i is assigned to node j

(denoted by V_i for application i) and must be placed on a data node with sufficient storage space.

For fault-tolerance, an application may request multiple replicas (denoted by R_i for application i). Applications may also have certain hardware requirements, e.g., type of storage device or presence of a particular CPU or GPU on the datanode. We call this requirement, node compatibility. An application i is compatible with node j if j meets all its hardware requirements ($C_{ij} = 1$).

Finally, applications may have data collocation restrictions. If an application x cannot share a node with y due to privacy or regulatory requirements, we set binary variable $U_{xy} = 1$.

B. Problem Formulation

Table I provides description of all the variables used in the equations below. The objective of our optimization model is to minimize the number of nodes while respecting all the user constraints and requirements.

$$minimize \sum_{j \in D} P_j \tag{1a}$$

subject to

$$\sum_{j \in D} B_{ij} = R_i \quad \forall \ i \in A \tag{1b}$$

$$\sum_{j \in D} B_{ij} \cdot V_i \leq T_j \quad \forall \ j \in D \tag{1c}$$

$$B_{ij} \leq C_{ij} \quad \forall \ i \in A, j \in D \tag{1d}$$

$$B_{xj} + U_{xy} \cdot B_{yj} \leq 1 \quad \forall \ x \in A, y \in A, j \in D \tag{1e}$$

$$P_j \geq B_{ij} \quad \forall \ i \in A, j \in D \tag{1f}$$

where, P_j is a binary variable that indicates whether node j is active or not.

Constraint 1b ensures that the required number of replicas for each application are created on separate datanodes. Constraint 1c guarantees that a datanode does not allocate more storage space than it has. Constraint 1d restricts allocation of applications to only the compatible datanodes. Constraint 1e ensures that data of applications that cannot be collocated is placed on separate nodes. Constraint 1f marks a datanode as active if an application is assigned to it.

Algorithm 1: Data Placement Heuristics.

```
Input: $Application\ (X)$, $Request$, $Current\_State$
Output: $New\_State$
if Request == "Add_Application" then
    $New\_State = ADD\_APPLICATION(X,\ Current\_State)$
else if Request == "Remove_Application" then
    $New\_State = Current\_State.remove(X)$
else if Request == "Add_Constraint" then
    if X cannot be served by its current node then
        $Current\_State = Current\_State.remove(X)$
        $ADD\_APPLICATION(X,\ Current\_State)$
    else
        $New\_State = Current\_State$ # No action.
else if Request == "Remove_Constraint" then
    $New\_State = Current\_State$ # No action.
```

Time scale of optimization: Our optimization problem is an integer linear program (ILP) with binary decision variables. Hence the problem is NP-complete [21]. However, when solving the problem in real-time, the scale of the system, e.g., number of datanodes or application requirements, may lead to slow computation, possibly impractical for deployment in a dynamic environment. To tackle such environments, heuristics can be applied to reduce the computation time.

Another variable is the cost of data movement, which is an expensive operation [22] and may lead to an unstable or unprofitable system. One approach is to add a penalty term in the objective (Equation 1a) and factor in the current node assignment of applications. As this solution will lead to higher computation times, we instead define heuristics that make informed decisions when responding to state changes.

C. Heuristics Algorithm

When a state changes in the storage system, a decision has to be made by the placement algorithm. State changes are categorized as, adding an application to the system, adding constraints to applications, removing or relaxing constraints, and removing an application. Furthermore, the algorithm could either ignore the change, modify the state with a good feasible action, or find the optimal solution.

Our proposed heuristic (Algorithm 1) works by characterizing the change, exploring the type of decision to apply, and finding the best way to apply that decision. Moreover, we run the optimization model periodically or after a certain number of changes to bring the system back to optimality.

Below we use the toy example from Figure 2 to describe how the algorithm handles various changes and applies different actions. Note that a combination of changes, e.g., adding multiple applications or constraints, is handled by solving them sequentially using the steps explained below.

1) Adding a New Application: Algorithm 2 presents the heuristics involved in calculating the new state.

Algorithm 2: Heuristics to Add an Application.

```
Input: $Application\ (X)$, $Current\_State$
Output: $New\_State$

$F \leftarrow List\ of\ Available\ Datanodes$
Remove nodes from F that fail sanity checks.
if X requested an isolated node then
    $F \leftarrow F - Active\_Nodes.$;
for $j \in F$ do
    if X has privacy conflicts with applications on j
      then $F \leftarrow F - j$;
    if X has hardware conflicts with j then $F \leftarrow F - j$;
if $|F| == R_x$ then  Assign X to F;
    $New\_State = Current\_State.add(X)$ ;
if $|F| - |Inactive\_Nodes| \geq R_j$ then
    $F \leftarrow F - Inactive\_Nodes$;
Sort F by amount of storage space available.
$j \leftarrow$ The first node with sufficient storage space for X.
Assign $F[j, j + R_x]$ to X.
    $New\_State = Current\_State.add(X)$
```

Modify the state: We first find a list of compatible nodes with enough storage space and not serving an unshareable application. The list is further narrowed down by excluding inactive nodes, unless all feasible nodes are inactive. Finally, the nodes are sorted by amount of available space and the one with lowest but sufficient space is selected.

For example in Figure 2, assume 2c to be the current state and a new application (App 5) with organization X's data needs to be added. Node B is excluded as it conflicts the privacy constraints. Nodes C and D are excluded as they are inactive. Node A is thus chosen to serve Application 5. The state of the system is modified accordingly.

Note that a chosen solution may be sub-optimal and moving some applications may lead to optimality. However, this is avoided to reduce the data movement cost (Section IV-B).

Optimize the state: Executing the optimization model is deferred, unless the heuristics cannot find a feasible solution.

2) Adding a Constraint:

Ignore the change: In some cases, it might be possible to not react to a new constraint when the change does not invalidate the current solution. For example, in Figure 2, assume (2a) is the current state and a new constraint is added that prohibits data of application X to be placed on the same node as application Y. This requirement does not invalidate the current placement and is ignored.

Modify the state: If adding a constraint to an application invalidates its current placement, we recalculate its placement. This is done by first performing the application removal and then application addition steps to determine another node. The removing here refers to logical steps taken by the data placement algorithm and the application is not physically moved in the storage system until the final decision is reached.

For example, in Figure 2, assume (2b) is the current state and a new constraint is added such that Application 4 cannot be placed on Node D. As this invalidates the current solution, Application 4 is first removed and Node D is set as inactive. Node A and D are excluded due to constraints. Because node C is inactive, and node B is already serving an application, node B is chosen for Application 4.

Optimize the state: Executing the optimization model is deferred, unless the heuristics cannot find a feasible solution.

3) Removing an Application or Constraints:

Ignore the change: As ignoring an application removal will not invalidate the current state, this is always the chosen course of action. Note that the system state may become sub-optimal but to avoid frequent data movement, such actions are deferred until the next periodic run of the optimization model.

V. Prototype Implementation

This section describes our prototype implementation of `CATER`. Note that a storage system that intends to use an external placement framework, such as `CATER`, must be able to communicate with the framework e.g., through API calls. Hence, a few modifications may be required for the storage system if its incapable of making such calls. We choose Apache Ozone for integration with `CATER` and describe the necessary modifications. In this section, we also explain the implementation of the data placement framework and the placement algorithm.

A. Ozone Modification

We opted for Apache Ozone as it is a distributed object store that can manage both small and large files alike. Unlike HDFS, Ozone separates namespace management and block space management. The namespace is managed by a daemon called the Ozone Manager (OM) whereas the Storage Container Manager (SCM) manages the block space. Ozone makes a data placement decision when it receives a request to store a file or object. By default, Ozone randomly selects nodes for data placement. We modify this behaviour to implement `CATER` by calling an external API (Section V-B). The API call returns the list of datanodes on which the application's data can be placed to satisfy the specified constraints. An external API approach allows re-use of this component when integrating with other data stores.

The Ozone implementation details are depicted in Figure 4. The OM component receives requests to store files. OM forwards the request to the SCM as a block allocation request. The SCM forwards this request to the Block Manager (BM) component [9]. We modified the BM to invoke the external placement API which returns a list of nodes to be used for placement. The BM uses this list to send a file-creation request to the Container Manager (CM). The CM creates the blocks on the selected nodes to store the files.

Occasionally Ozone might change the location of the file. For example, because of the recommendation of the load balancer component, it may change the locations of files to maintain a balanced state. Such activities are handled by an alert to `CATER` via the API and receiving updated instructions. `CATER` may also be given an up-to-date map of the desired allocation to be considered as an input.

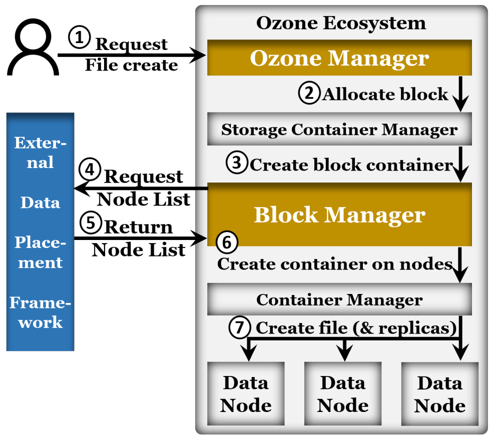


Fig. 4. Ozone Internal Working and Calling Placement Framework.

B. Placement API Implementation

CATER is implemented as an external REST API. API requests detail the required action and pass on information on the current node state. The API obtains information on application constraints via a database (key-value store). Using the current state and the constraints, the framework creates the list of the nodes for the request.

The framework supports stateless and stateful modes for maintaining the system state. In stateless mode the framework expects the storage system to maintain the current state and pass it along with each request. Whereas in stateful mode, the framework maintains the current allocation state itself. The trade-off among the two modes is between bandwidth and instantaneous access to up-to-date state. A storage provider may choose a mode based on their preferences.

The API requires an application server such as Apache Tomcat [23]. Our prototype stores the information in a key-value store. This key-value store keeps a record of constraints for all applications and the current application placement on different nodes. The Placement API uses these two pieces of information to create a list of nodes for the allocation of the current request.

The placement algorithm (Section V-C) is implemented in Python. The data placement API framework is written in Flask, which is a micro web framework written in Python. An example call to the *ADD_APPLICATION* function of heuristics as given in Algorithm 1 looks like the pseudo-code fragment in Listing 1.

```
endpoint = 'placement.api:80/perform'
placement_request = { 'application': '2', 'operation': 'Add_Application',
 'node_state': {
  'a': {'total_space':400, 'available':200, 'hardware':'x86', 'apps':['1']},
  'b': {'total_space':200, 'available':200, 'hardware':'ARM', 'apps':['']},
  'c': {'total_space':200, 'available':200, 'hardware':'x86', 'apps':['']},
 }
}
response = requests.post(endpoint, json=placement_request)
```

Listing 1. Example Call to Placement API in Stateless Mode.

The API also supports the other three operations discussed in Algorithm 1 i.e. *Remove_Application*, *Add_Constraint*, *Remove_Constraint*. The API supports both GET and POST HTTP methods. The POST method handles the stateless mode and expects the current state of the nodes along with other inputs while the GET method handles the stateful mode. The response to each API call is an updated node state, similar to the one shown in Listing 1.

C. Placement Algorithm Implementation

After processing the REST API calls, the data placement framework forwards the request along with the current state to the placement algorithm which in our case is either the optimization model or CATER heuristics-based algorithm. Note that the modular structure of the proposed solution along with common API definitions allows for easily updating or replacing the placement logic without making any changes to the overall system. This approach also allows the usage of multiple placement algorithms running concurrently where a storage system and the platform provider can request a solution based on a particular approach.

The placement optimization model (Section IV-B) is implemented using OR-Tools, an open-source optimization library developed by Google [24]. The optimization model is an integer program, implemented in Python and solved using Constrained Programming with Satisfiability methods (CP-SAT) solver from OR-Tools.

CATER heuristics algorithm is also implemented in Python and solves the same problem with same input using the mechanism defined in Algorithm 1. In cases where CATER requires optimization, it calls the optimization module described above.

VI. Experimental Evaluation

To evaluate the performance of the proposed solution, we built an event-driven simulator and conducted various experiments. We also integrated our prototype implementation with Apache Ozone (Section V) to validate and evaluate the proposed framework in a real-world deployment.

A. Simulation Setup

Our simulation setup and parameters are derived from the requirements of the BRAINE Edge platform and its industrial use case [25]. The platform consists of cutomizeable, small form-factor EMDCs, specifically designed for Edge environments. The design is modular, with x86 CPU's, ARM CPU's, FPGA's, GPU's, and NvME storage available to populate each system slot. The EMDC backplane connector includes both a PCIe interface and 25G Ethernet links to connect system slots together, and also connect to the edge network. [26]. The BRAINE EMDC supports anywhere from 8 to 24 slots, which influences the number of nodes we will define to evaluate CATER's performance. To meet the requirements of the factory use-case, and to generate diverse meaningful evaluation results, we assume six ARM and six x86 nodes.

We consider 50 applications of varying sizes with 30 hardware-agnostic applications and 10 of them requiring an ARM node and x86 node each. These applications represent workloads running as docker containers (or Kubernetes pods) in the BRAINE EMDC, with each docker image either complied for x86, or ARM, or both processor types. In BRAINE's

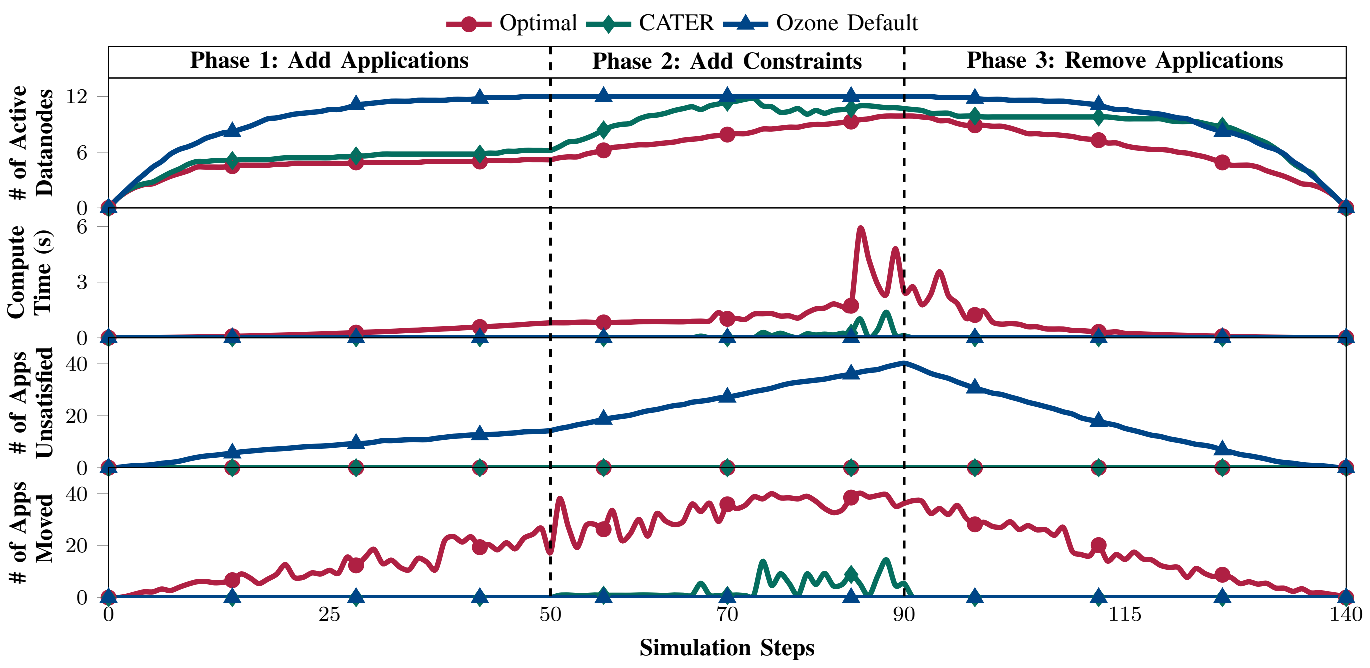


Fig. 5. Simulation results averaged over multiple runs. Unlike Ozone default, CATER and Optimal satisfy all application requirements. In comparison to Optimal, CATER uses more nodes but improves computation time and application movement.

industrial use case for example, each robot on an assembly line has its own container, and storage requirements. The application and storage requirements of each robot may differ, thus a set of 50 applications with a subset of them with certain placement constraints is similar to the scenario encountered in the BRAINE use case [25].

For collocation constraints, we consider three classes, low-constrained, medium-constrained, and high-constrained applications that can be collocated with 80%, 70%, and 50% of all applications respectively. These numbers are chosen to simulate and analyze various scenarios that may arise in a real-world deployment.

To demonstrate the impact of various system changes (Section IV-C), we start from a blank state with no active applications and conduct the experiments in three phases:

Adding applications: In Phase 1, we sequentially add all 50 applications to the environment while assuming that 10 out of 50 applications have collocation constraints.

Adding constraints: In Phase 2, we evaluate a highly-constrained system by sequentially increasing the number of applications with collocation constraints from 10 to 50.

Removing applications: In Phase 3, we sequentially reduce applications until all have left the system.

We repeat the above experiment 10 times by varying application specifications using weighted uniform distributions and compare three algorithms.

- **Optimal**: For reference, we solve the optimization model (Section IV-B) for each step in the experiment.
- **`CATER`**: Solves the placement problem using the proposed heuristics (Section IV-C) and when the heuristics fails, it runs the optimization model to get feasible and optimal solutions.
- **Default**: A default Ozone placement algorithm without privacy handling functionality. Allocates nodes considering storage space requirements of applications.

We compare these approaches using the following performance metrics:

- **Nodes used**: Number of active nodes that have one or more applications placed on them at a given time.
- **Unsatisfied applications**: Number of applications that were placed on a datanode that violates any of their requirements e.g., hardware or collocation constraints.
- **Computation time**: Amount of time taken to solve the placement problem.
- **Movements**: The number of times applications were moved from one datanode to another.

The simulations are carried out on a Linux virtual machine with 2.2GHz 8-core Intel CPU and 64GB RAM. The results are presented and analyzed in the following section.

B. Simulation Results

Figure 5 shows the results of our experiments, comparing `CATER` with the optimal result and the default Ozone solution. As a general observation, we can see that `CATER` and `Optimal` both managed to respect all of the application constraints and they did so while using fewer storage nodes than Ozone default. `CATER` outperformed `Optimal` in regards to execution time, but also more significantly, saw a substantial reduction in the number of data movements, while satisfying all application constraints.

Inspecting the number of nodes used, we see that Default always allocated most nodes, due of its lack of a specific objective other than capacity. In fact it occupied all available storage nodes for 32% of the experiment duration. In contrast,

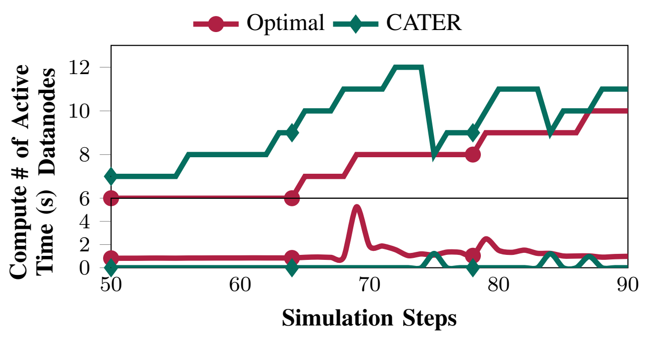


Fig. 6. CATER vs Optimal. Analyzing optimization needs of CATER.

both CATER and Optimal minimised the number of nodes used. Optimal fared best overall, at the cost of moving data between nodes at each step in comparison to CATER.

When CATER was unable to find a solution it invoked Optimal, evident from the decrease in nodes used between steps 70-80. This manifested in corresponding peaks between movements and computation time, with matching reduction in datanodes used. Figure 6 shows this for a single experiment, evident in which are the aligned changes in number of nodes and computation time.

Quantifying the cost of data movement between storage nodes, we captured the differences between the three approaches. As expected, the Default did not move any applications. Optimal exceeded CATER in data movements by a factor of 19 which also resulted in 19 times higher CPU execution. Evident from the plots is that the problem complexity is at its peak towards the end of Phase 2 due to the combination of the number of applications and constraints. As Optimal is designed to minimize the number of active nodes, it searches the whole problem space and tries to find any possible combination which can reduce the active nodes. On the other hand, CATER focuses on finding a good feasible solution based on the current system state. Consequently, CATER results in fewer data movements and lower computational cost.

As the constraints are being removed in Phase 3, we notice that the advantage of Optimal over CATER in terms of the number of occupied nodes increased. This is because CATER did not see fit to make any alterations since all the constraints were still being satisfied.

One can conclude from the results that CATER is an attractive choice when seeking to respect application constraints, even when the set of applications and constraints is highly dynamic. Using the optimal approach resulted in a substantially higher number of movements which is detrimental to an Edge storage system and if not handled carefully can lead to delays for applications in accessing their data. Over the experiment lifetime, Optimal occupied just 1.6 fewer storage nodes compared to CATER, reinforcing the clear benefit of CATER as an efficient solution for constraint-based edge storage.

C. Scalability Analysis

We note that CATER calls on the optimization model only when it is not possible to accommodate a new request using its heuristic. In fact, the processing time for the optimization model is quite fast, as this subsection addresses empirically. During these experiments, we fixed the number of nodes to 24 as referenced in an Edge project as being typical for an EMDC [20]. We evaluated up to 240 highly-constrained applications (as defined in Section VI-A) and varied applications with collocation constraints from 10% to 50%.

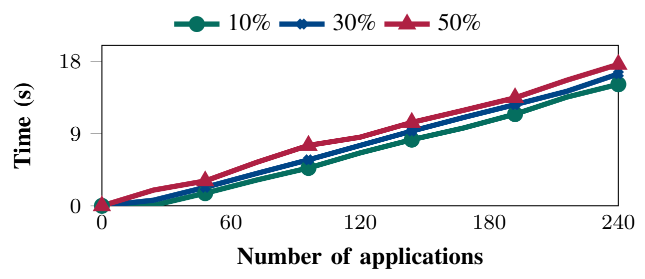


Fig. 7. Scalability analysis of optimization model.

The results are depicted in Figure 7, based on an average of five runs. We observe that even for the most-constrained case and 240 applications, the optimization model solves the problem in less than 18 seconds. Given that CATER calls the optimization model quite infrequently, we posit that the computation time is not significant enough to hinder the scalability of our solution in practical deployments.

D. Prototype Implementation Results

We perform these experiments using our modified Apache Ozone deployment that interacts with the placement framework. With these experiments we aim to demonstrate the feasibility of the proposed framework and evaluate the algorithms in a real-world deployment. We created up to 40 applications in same ratio of constraints as discussed in Section VI-A. and compared Optimal and CATER, based on the the CPU utilization and API response time. The response time is the time elapsed from when Ozone invokes call a framework API until it receives the response.

The experiments were performed on a machine with 16-core CPU and 64GB RAM, with 8 containers running an Ozone datanode each. In each experiment, a different number of total applications is added to the system and the performance metrics are gathered using Linux utility *top* and *time* functions. Table II presents the results for each experiment.

We observed that, CATER consumed lower CPU than Optimal to perform the same operations. This is due to the higher computational requirements of Optimal. The response time of both approaches grew quite considerably as the load on the system increased, but with a markedly lower value for CATER. Note that in accordance with the simulation results (Figure 5), both Optimal and CATER satisfied all the application constraints.

To analyze the response times further, we looked at one particular experiment where a total of 20 applications were added to the system. Table III shows the API response time for each application separately. Results show that the response time for Optimal increased with the application number.

TABLE II
CATER VS. OPTIMAL PERFORMANCE IN REAL DEPLOYMENT ON OZONE.

Metric	Algo.	Total Added Applications				
		1	**5**	**10**	**20**	**40**
% CPU Used	CATER	20.5	35.0	70.2	74.3	79.5
	OPTIM	27.2	45.1	80.4	84.3	93.5
Response-Time (s)	CATER	0.27	0.85	1.56	2.99	5.80
	OPTIM	0.28	0.87	1.70	3.86	9.80

TABLE III
CATER VS. OPTIMAL API RESPONSE TIME PER APPLICATION ADDED.

App #	CATER	OPTIM	App #	CATER	OPTIM
1	0.14s	0.14s	**11**	0.14s	0.17s
2	0.13s	0.14s	**12**	0.14s	0.19s
3	0.14s	0.14s	**13**	0.14s	0.19s
4	0.14s	0.15s	**14**	0.15s	0.20s
5	0.14s	0.15s	**15**	0.14s	0.21s
6	0.15s	0.16s	**16**	0.14s	0.21s
7	0.14s	0.16s	**17**	0.15s	0.22s
8	0.14s	0.16s	**18**	0.14s	0.23s
9	0.14s	0.17s	**19**	0.14s	0.24s
10	0.14s	0.17s	**20**	0.15s	0.26s

This is due to the fact that as the nodes are populated with applications, it increases the complexity and the search space for `Optimal` as it tries to minimize the objective function and meet all constraints (Section IV-B). Whereas, for `CATER` the response time remains fairly consistent for each application.

VII. Conclusion

Emerging data-intensive applications and diverse resource environments are expected to induce constraints on where application data is stored. These restrictions may be privacy-driven, regulatory, or related to hardware compatibility. Current storage solutions focus on robust and highly-efficient storage but for the most part neglect application-level constraints. In this paper we presented `CATER` as an external service that runs optimization models or heuristic algorithms. We evaluated our solution through simulation as well as live Apache Ozone deployment. We demonstrated that our proposed approach is able to respect constraints and quantify the cost in terms of resources and performance. CATER used 23% fewer datanodes on average when compared to the default Ozone algorithm, while satisfying all constraints and with only a modest amount of data movement between nodes.

Acknowledgment

This project has received funding from ECSEL-JU under grant No 876967, KDT-JU under grant No 10109756 and Science Foundation Ireland under grant No 13/RC/2077_P2.

References

[1] I. F. Adams, N. Agrawal, and M. P. Mesnier, "Enabling near-data processing in distributed object storage systems," in *Proceedings of the 13th ACM Workshop on Hot Topics in Storage and File Systems*, 2021.

[2] S. Mazumdar, D. Seybold, K. Kritikos, and et al., "A survey on data storage and placement methodologies for cloud-big data ecosystem," *Journal of Big Data*, vol. 6, no. 15, 2019.

[3] H. Li, L. Yu, and W. He, "The impact of GDPR on global technology development," 2019.

[4] K. Narayana and K. Jayashree, "Survey on cross virtual machine side channel attack detection and properties of cloud computing as sustainable material," *Materials Today: Proceedings*, vol. 45, 2021.

[5] C.-T. Wang and C.-S. Chiu, "Competitive strategies for Taiwan's semiconductor industry in a new world economy," *Technology in Society*, vol. 36, 2014.

[6] P. Szántó, T. Kiss, and K. J. Sipos, "Energy-efficient AI at the edge," in *2022 11th Mediterranean Conference on Embedded Computing (MECO)*, 2022.

[7] S.-Q. Long, Y.-L. Zhao, and W. Chen, "Morm: A multi-objective optimized replication management strategy for cloud storage cluster," *Journal of Systems Architecture*, vol. 60, no. 2, 2014.

[8] P. J. Stuckey, "Lazy clause generation: Combining the power of SAT and CP (and MIP?) solving," in *International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) Techniques in Constraint Programming*. Springer, 2010.

[9] Apache Ozone, "Apache ozone documentation," 2022, Accessed 25 Aug 2022. [Online]. Available: https://ozone.apache.org/docs/1.1.0/index.html

[10] A. Kaur, P. Gupta, M. Singh, and A. Nayyar, "Data placement in era of cloud computing: a survey, taxonomy and open research issues," *Scalable Computing: Practice and Experience*, vol. 20, no. 2, pp. 377–398, 2019.

[11] M. Liu, L. Pan, and S. Liu, "Cost optimization for cloud storage from user perspectives: Recent advances, taxonomy, and survey," *ACM Computing Surveys*, 2023.

[12] T. T. Ke and K. Sudhir, "Privacy rights and data security: GDPR and personal data markets," *Management Science*, vol. 69, no. 8, pp. 4389–4412, 2023.

[13] C. Xu, Y. Qu, Y. Xiang, and L. Gao, "Asynchronous federated learning on heterogeneous devices: A survey," *Computer Science Review*, vol. 50, p. 100595, 2023.

[14] M. Y. Eltabakh and et al., "Cohadoop: flexible data placement and its exploitation in hadoop," *Proceedings of the VLDB Endowment*, vol. 4, no. 9, 2011.

[15] J. Dharanipragada and et al., "Tula: A disk latency aware balancing and block placement strategy for hadoop," in *2017 IEEE International Conference on Big Data (Big Data)*. IEEE, 2017.

[16] Z. Liu, W. Hua, X. Liu, D. Liang, Y. Zhao, and M. Shi, "An efficient group-based replica placement policy for large-scale geospatial 3D raster data on hadoop," *Sensors*, vol. 21, no. 23, 2021.

[17] R. Padmanaban and R. Mukesh, "Hadoopsec: Sensitivity-aware secure data placement strategy for big data/hadoop platform using prescriptive analytics," *GSTF Journal on Computing (JoC)*, vol. 5, no. 3, 2020.

[18] P. Revathy and R. Mukesh, "Hadoopsec 2.0: Prescriptive analytics-based multi-model sensitivity-aware constraints centric block placement strategy for hadoop," *Journal of Intelligent & Fuzzy Systems*, vol. 39, no. 6, 2020.

[19] A. K. TK, H. Liu, J. P. Thomas, and G. Mylavarapu, "Identifying sensitive data items within Hadoop," in *2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems*. IEEE, 2015.

[20] J. J. V. Olmos, F. Cugini, F. Buining, N. O'Mahony, T. Truong, L. Liss, T. Oved, Z. Binshtock, and D. Goldenberg, "Big data processing and artificial intelligence at the network edge," in *2020 22nd International Conference on Transparent Optical Networks (ICTON)*, 2020.

[21] S. A. Cook, "The complexity of theorem-proving procedures," *Proceedings of the third annual ACM symposium on Theory of computing*, 1971.

[22] D. C. Marinescu, *Cloud computing: theory and practice*. Morgan Kaufmann, 2022.

[23] Apache Tomcat Team Members, "Apache tomcat application server," 2023, Accessed 1 Mar 2023. [Online]. Available: https://tomcat.apache.org/

[24] Google, "OR-Tools Open-source Optimization Library," 2022, Accessed 29 Aug 2022. [Online]. Available: https://github.com/google/or-tools

[25] S. Ahearne, A. Khalid, M. Ron, and P. Burget, "An AI factory digital twin deployed within a high performance edge architecture," in *2023 IEEE 31st International Conference on Network Protocols (ICNP), Workshop on Cloud-Edge Continuum*, 2023.

[26] BRAINE Project Team Members, "D2.1 first project report on the status of WP2," 2022, Accessed 28 Apr 2023. [Online]. Available: https://www.braine-project.eu/deliverables/