Cluster Computing (2026) 29:698
https://doi.org/10.1007/s10586-026-06520-z

Failure-resilient and carbon-efficient deployment of microservices over the cloud-edge continuum

Francisco Ponce[1,4] · Simone Gazza[2] · Andrea D'Iapico[3] · Roberto Amadini[2] · Antonio Brogi[1] · Stefano Forti[1] · Saverio Giallorenzo[2] · Pierluigi Plebani[3] · Davide Usai[1] · Monica Vitali[3] · Gianluigi Zavattaro[2] · Jacopo Soldani[1]

Received: 31 October 2025 / Revised: 15 May 2026 / Accepted: 10 August 2026

Abstract
Deploying microservice-based applications (MSAs) on heterogeneous and dynamic Cloud-Edge infrastructures requires balancing conflicting objectives, such as failure resilience, performance, and environmental sustainability. In this article, we introduce the FREEDA toolchain, designed to automate the failure-resilient and carbon-efficient deployment of MSAs over the Cloud-Edge Continuum. The FREEDA toolchain continuously adapts deployment configurations to changing operational conditions, resource availability, and sustainability constraints, aiming to maintain service continuity while reducing carbon emissions. We also introduce an experimental suite using diverse simulated and emulated scenarios to validate the effectiveness of the toolchain against real-world challenges, including resource exhaustion, node failures, and carbon intensity fluctuations. The results demonstrate FREEDA's capability to autonomously reconfigure deployments by migrating services, adjusting flavour selections, or rebalancing workloads, successfully achieving an optimal balance among resilience, efficiency, and environmental impact.

Keywords Cloud-edge continuum · Microservices · Failure resilience · Environmental sustainability

1 Introduction

Modern applications are increasingly delivered as *microservice-based architectures* (MSAs), namely by decomposing their functionalities into loosely coupled, independently deployable services. MSAs have become the de-facto standard for cloud-native systems, as they enable scalability, modularity, and continuous delivery. However, the deployment of MSAs is becoming significantly more challenging due to the ongoing evolution of computing infrastructures.

In particular, the proliferation of IoT devices and latency-sensitive applications is driving the transition from centralized cloud computing towards the *Cloud-Edge computing continuum* [1, 2]. Cloud-IoT infrastructures span from large cloud data centers to edge nodes and resource-constrained devices. Such environments are inherently *heterogeneous* (due to, e.g., diverse hardware and software stacks) and *dynamic* (due to, e.g., node churn and failures). These characteristics complicate the deployment of MSAs, as the microservices in an MSA must be distributed across Cloud-IoT infrastructures while enforcing functional and non-functional requirements.

Non-functional requirements aim at preserving the Quality of Service (QoS) of an MSA, through e.g., latency, availability, or bandwidth requirements. QoS requirements must be enforced when operating MSAs across the Cloud-Edge infrastructures, with *failure resilience* being key to avoid microservice disruptions and/or cascading degradations across interdependent microservices.

At the same time, there is a growing need to account for the *environmental sustainability* of ICT. Initiatives such as the EU strategy of "building a climate-neutral, green, fair, and social Europe" [3], which promotes the environmentally sustainable growth of European industries, including IT [4], reflect this broader trend. In this context, a key concept is carbon-awareness [5]. This is typically based on the

✉ Jacopo Soldani
jacopo.soldani@unipi.it

1 University of Pisa, Pisa, Italy

2 University of Bologna, Bologna, Italy

3 Politecnico di Milano, Milano, Italy

4 Pontificia Universidad Católica de Valparaíso, Valparaíso, Chile

Published online: 26 August 2026

carbon intensity of running software, defined as the amount of carbon emissions (expressed in grams of CO_2-equivalent per kWh) associated with the power consumed by the node running such software [6]. Carbon intensity varies across geographical locations and over time, depending on the energy mix powering an infrastructure node, e.g., renewable energy vs. fossil sources. Therefore, deployment decisions in the Cloud-Edge continuum can significantly affect the carbon footprint of MSAs.

However, optimizing for environmental sustainability introduces the need for trade-offs with failure resilience requirements. For instance, carbon emissions could be reduced by consolidating microservices onto a small number of low-carbon nodes, while improving failure resilience would require replicating the same microservices across multiple heterogeneous nodes. Such conflicting objectives highlight the need for multi-criteria deployment strategies, which explicitly account for suitable trade-off among environmental sustainability and failure resilience.

1.1 Research objective

The main objective of this work can be stated as follows: *How can we deploy MSAs over Cloud–Edge infrastructures while jointly ensuring failure resilience and carbon efficiency?* More precisely, our objective is to devise a support for deploying MSAs over Cloud-Edge infrastructures that (i) explicitly enforces the resilience to node and service failures, and (ii) explicitly considers the carbon intensity of infrastructure nodes for limiting the carbon emissions of deployed MSAs.

Additionally, in line with our previous work [7], we consider the microservices forming an MSA to be available in multiple versions, called *flavours*. Each flavour of a microservice provides akin functionality, but with different execution times and QoS. For instance, a microservice performing a classification task could be implemented in two different flavours, i.e., *high*, taking 10 s on average to produce results with 3% error, and *low*, taking 1 s on average to produce results with 6% error. While the *high* flavour would be preferable, yielding it to better results, it also consumes more energy for running, and it will probably also require more computing resources. With this in mind, our main goal is to answer the above research question by selecting suitable microservice flavours and determining their placement across heterogeneous and dynamic Cloud–Edge infrastructures.

1.2 Limitations of existing approaches

Various works target reducing energy consumption or carbon emissions of software applications [6]. A number of these works focus on applications deployed over either Edge infrastructures, e.g., [8–14], or Cloud platforms, e.g., [15–17], hence witnessing the growing interest and need for environmental sustainability. At the same time, only a few works targeted the deployment of multi-service applications (such as MSAs) over Cloud-Edge infrastructures, typically focusing on different objectives than ours [18]. For example, [19] proposes deploying multi-flavoured services according to operator preferences and cost objectives, employing a greedy strategy to minimize operational expenses. Yet, their approach does not incorporate sustainability or carbon awareness. Other works, e.g., [20], exploit flavours in the context of MSAs to adapt workflows and mitigate environmental impact, but they do not address deployment challenges in heterogeneous, distributed infrastructures. As a result, the state of the art still lacks solutions for planning the failure-resilient and carbon-efficient deployment of MSAs over Cloud–Edge infrastructures, which is precisely our research objective.

1.3 Contributions

To address our research objective, we introduce the *FREEDA toolchain* to plan the failure-resilient and carbon-efficient deployment of MSAs across Cloud-Edge infrastructures. FREEDA automates the selection of microservice flavours and plans their placement on Cloud-Edge infrastructure nodes, by also enabling to reconfigure an MSA deployment to adapt to, e.g., changing deployment requirements, operational conditions, or resource availability. Unlike cluster-level schedulers such as Kubernetes [21], which operate on individual resources, FREEDA reasons at the application-level across the whole available Cloud-Edge infrastructures.

We also report on *controlled experiments* assessing the effectiveness of FREEDA. The experiments consist of heterogeneous realistic scenarios simulating or emulating the use of FREEDA to deploy MSAs across Cloud-Edge infrastructures, also in the presence of e.g., resource exhaustion, node failures, or carbon intensity fluctuations. The results show that FREEDA can effectively adapt MSA deployments by selecting appropriate flavours and redistributing services, achieving a balanced trade-off between resilience to failures and carbon efficiency, while also avoiding unnecessary reconfigurations.

The remainder of this article is structured as follows. Section 2 retakes and extends our constraint-based deployment model. Section 3 provides an overview of the FREEDA toolchain. Section 4 presents and discusses the simulation results, while Sect. 5 focuses on the emulation results. Finally, Sects. 7 and 8 review related work and provide concluding remarks, respectively.

This paper extends our previous works [7, 22], where we first presented our constraint model [7] and the ideas behind FREEDA's failure enhancer [22]. This article extends the deployment model (Sect. 2) and provides a thorough description of the full FREEDA toolchain (Sect. 3). Additionally, this article provides brand-new experiments (Sects. 4 and 5) to assess how effectively FREEDA can support the sustainable and failure-resilient deployment of MSAs over Cloud-Edge infrastructures.

2 Deployment model

2.1 The FREEDA model

In our previous work [7], we introduced a constraint optimisation model that provides the mathematical foundation of the FREEDA framework. The goal of this model is to transform high-level deployment specifications – concerning application components, their flavours, deployment requirements, and infrastructure characteristics – into a rigorous optimisation problem whose solutions correspond to feasible deployments. Specifically, the model jointly: (i) selects each component and its respective flavour and assigns it to a node, (ii) ensures compliance with all deployment requirements expressed in the FREEDA YAML specification,[1] including energy and carbon budgets, and (iii) prioritises the deployment of the most powerful flavours whenever possible, in line with application owners' preferences. The full formal specification of the model is articulated into four essential parts, i.e., parameters, variables, constraints, and the objective function, which can be found in our previous work [7] and are recapped in Appendix 9.. Such parts are also described informally in the following paragraphs.

Parameters provide the input data for each deployment instance, capturing the set of application components and their associated flavours, dependencies between components, resource requirements (both consumable, such as CPU, RAM, and storage, and non-consumable, such as availability or security), infrastructure specifications (e.g., node capacities, link latency and availability, and per-resource monetary and carbon costs), and budget thresholds for expenditure and emissions. Each flavour has an *importance* value, which is defined statically to prioritize more powerful flavours: the higher the importance value, the more powerful the flavour.

Variables encode the deployment decisions as binary indicators $D^c_{i,j} \in \{0, 1\}$ where c, i, and j respectively denote a component, a flavour (of c), a node in the infrastructure. The constraint solver sets $D^c_{i,j} = 1$ if and only if component c is deployed in flavour i on node j. This representation is solver-agnostic; in fact, various solving technologies can execute the model, including constraint programming, mixed-integer linear programming, or boolean satisfiability.

Constraints formalise the validity conditions of a deployment (cf. Appendix 9.). They guarantee, e.g., that each component is deployed in at most one flavour on one node, that mandatory components are always deployed, that required dependencies are satisfied, and that non-essential components are never deployed in isolation. Resource constraints ensure that aggregate consumptions do not exceed node capacities, while network link constraints ensure required latency and availability requirements. Budget constraints enforce compliance with financial and carbon limitations.

The objective function maximizes the importance of deployed flavours rather than directly minimizing costs or emissions, avoiding "empty deployments" where no component is deployed to reduce expenses. Budgets are enforced as hard constraints, while the optimization prioritizes higher-importance flavours wherever feasible.[2]

2.2 Minimizing deployment changes

We here extend [7] by focussing on reducing *changes* across successive deployments. Indeed, while adapting MSAs to changing conditions (e.g., monetary and energy budgets, or software and infrastructure failures), minimising the number of changes across successive deployments is critical for maintaining overall system availability, performance, and stability. This additional objective introduces a new minimisation criterion for the solver, i.e., when a new deployment is requested, the solver is instructed to minimise the number of components that either switch flavour or are relocated to a different node w.r.t. the current deployment. We capture this property by tracking each currently deployed component c, i.e., those for which $D^c_{i,j} = 1$ for some flavour i and node j. We aim at preserving their deployment in the subsequent configuration whenever possible. In other words, if $D^c_{i,j} = 1$ in the current deployment, the solver attempts to enforce $D^c_{i,j} = 1$ in the new deployment as well.

Formally, this requirement corresponds to maximizing the number of components that remain deployed with the same flavour on the same node, that is:

$$\max \sum_{\bar{D}^c_{i,j}=1} D^c_{i,j} \tag{1}$$

where $\bar{D}$ denotes the already computed matrix of decision variable *values* from the current deployment, and D represents the matrix for the new deployment. Flavour changes

[1] https://github.com/FREEDA-Project

[2] A full example of MSA deployment with the FREEDA deployment model can be found in our previous work [7].

and node relocations are treated equivalently because both operations ultimately require redeploying the component.

3 FREEDA toolchain

The FREEDA toolchain (Fig. 1) aims to address the demand for DevOps support in deploying an MSA A over a Cloud-Edge infrastructure I, guided by a set of deployment requirements R. The infrastructure description I includes information such as resource utilization costs, hardware and software capabilities of nodes, and their current load and availability. In contrast, the requirements R specify the needs of the services composing A, including hardware and software dependencies, network QoS, security, and failure resilience. Additionally, R may define deployment budgets, both monetary (i.e., the maximum affordable cost) and sustainability-related (i.e., the maximum permissible energy consumption and carbon emissions). FREEDA operates reactively by relying on the continuous monitoring of the deployed MSA and triggering adaptation whenever the monitored data witnesses failures or violations of R. When such events occur, FREEDAs computes a new deployment configuration that restores compliance with the specified requirements. At the same time, FREEDA incorporates a proactive dimension into its adaptation strategy by generating additional constraints derived from previously observed failures or violations, intending to reduce the likelihood that the same failures/violations happen again in future (re-) deployments.

The FREEDA toolchain generates a deployment plan D for A over I by holistically identifying a trade-off among the requirements in R. As shown in Fig. 1, the proposed solution is divided into two main phases (*Enrichment* and *Trade-Off*), each supported by two main components. The first phase enhances the failure resilience (Sect. 3.1) and environmental sustainability (Sect. 3.2) of the MSA by leveraging historical data H, which includes information from current and past deployments of A (e.g., logs) as well as monitoring data from I (e.g., node availability or load). This step produces an enriched set of requirements R+ that includes carbon-aware and failure-resiliency considerations.

In the second phase, the trade-off analysis tools (Sects. 3.3 and 3.4) process A and R+ to produce a deployment plan D together with an explanation E. When a solution is found, D represents a trade-off among the multiple – and potentially conflicting – deployment requirements. E instead explains the reasoning behind the enrichment process, which triggers the transformation of R+ into r+. This ensures that DevOps engineers are not only informed of the final deployment plan but also understand the rationale for any adjustments made to the application specification.

3.1 Failure enhancer

The Failure Enhancer is responsible for generating soft requirements[3] within R that aim to improve the resilience of microservices prior to deployment. These requirements help, for instance, to avoid deploying services on nodes known to fail under certain load conditions or on nodes whose failure is predicted to occur soon based on available historical data. As described in [22], the Failure Enhancer produces three types of constraints: *Affinity*, *Anti-affinity*, and *Avoid*. An *Affinity* constraint indicates that, if deploying components C and S in flavours FC and FS, respectively, they should be placed on the same node. An *Anti-affinity* constraint suggests avoiding placing the components C and S on the same node, when deployed in flavours FC and FS, respectively. Finally, an *Avoid* constraint states that component C, if deployed in flavour FC, should not be placed on node N. Figure 2 showcases the Prolog clauses used to generate the above-described soft constraints, which are discussed hereafter.

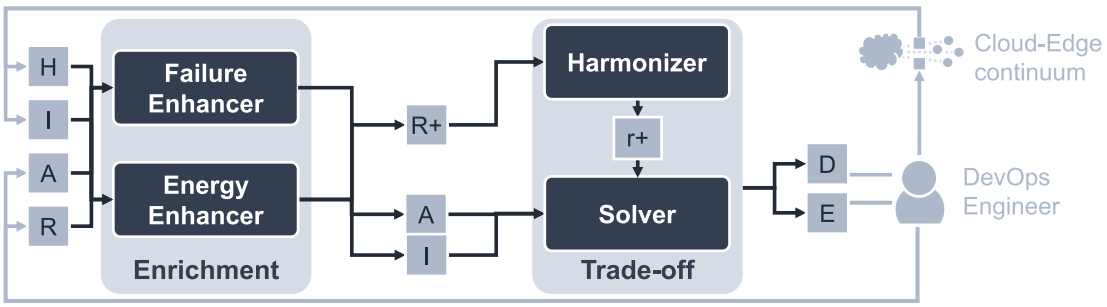

Fig. 1 A bird's-eye view of FREEDA's toolchain

[3] We hereafter refer to the requirements generated by the enhancers as *soft*, since FREEDA may decide to discard them upfront (if needed) *before* computing a feasible deployment. This is to distinguish soft requirements from those specified by DevOps engineers, which must instead be enforced.

```
1  suggested(affinity(d(C,FC),d(S,FS))) :-
2      deployedTo(C,FC,N), deployedTo(S,FS,M),
3      dif(C,S), dif(N,M),
4      timeoutEvent(C,S,T),
5      \+(
6          congested(N,M,T);
7          disconnected(N,T);
8          disconnected(M,T)
9      ).
10 suggested(avoid(d(C,FC),N)) :-
11     deployedTo(C,FC,N), deployedTo(S,_,M),
12     dif(C,S), dif(N,M),
13     timeoutEvent(C,S,T),
14     ( congested(N,M,T); disconnected(N,T) ).
15 suggested(antiaffinity(d(C,FC),d(S,FS))) :-
16     deployedTo(C,FC,N),
17     ( unreachable(C,T); internal(C,T) ),
18     overloaded(N,R,T), race(N,R,C,FC,S,FS,T).
19 suggested(avoid(d(C,FC),N)) :-
20     deployedTo(C,FC,N),
21     ( unreachable(C,T); internal(C,T) ),
22     (
23         (overloaded(N,_,T), \+race(N,_,C,FC,_,_,T));
24         disconnected(N,T)
25     ).
```

Fig. 2 Failure enhancer prolog clauses [22]

3.1.1 Knowledge representation

The current MSA deployment information is denoted via Prolog facts like `deployedTo(C,F,N)`, indicating that component `C` is deployed in its flavour `F` to node `N`. From an MSA failure perspective, the Failure Enhancer assumes that timeout events between components `C1` and `C2` are denoted via timestamped facts like `timeoutEvent(C1,C2,Timestamp)`. Besides, internal error and unreachability for a component `C` are denoted via facts like `internal(C,Timestamp)` and `unreachable(C,Timestamp)`, respectively.

On the other hand, considering infrastructure logging, the predicate `congested(N,M,T)` identifies that link congestion between nodes `N` and `M` occurred at time `T`. Likewise, the predicate `disconnected(N,T)` holds true if node `N` incurred a network disconnection at time `T`. Predicate `overloaded(N,R,T)` denotes the situation in which a specific resource `R` (e.g., RAM or CPU) was subject to overloading at time `T`. Last, predicate `race(N,R,C,FC,S,FS,T)` denotes that, at time `T`, components `C` (flavoured `FC`) and `S` (flavoured `FS`) were racing for resource `R` on the same node `N`.

3.1.2 Enhancing failure resilience

Based on the above, the Failure Enhancer generates a set of suggested soft constraints to improve the resilience of the current deployment by embedding rules of thumb to improve placement decisions. Indeed, it finds all distinct suggested constraints by checking which clauses of predicate `suggested/1` fire in the considered knowledge base. We here discuss the four clauses of such a predicate shown in Figure 2, noting, however, that they can be easily extended or refined to account for more MSA failures and/or network conditions.

The first clause of `suggested/1` identifies that an interaction between two `dif` ferent components `C` (flavoured `FC`) and `S` (flavoured `FS`), deployed to two distinct nodes `N` and `M` (lines 2–3), went through a timeout event (line 4), despite no congestion or disconnection events occurred (lines 5–9). To avoid this from happening again, the Failure Enhancer suggests deploying `C` and `S` closer, by adding an affinity constraint between the two components in their current flavours, i.e., `affinity(d(C,FC),d(S,SF))` (line 1).

The second clause of `suggested/1` identifies a timeout event at time `T` at component `C` in its flavour `FC` (line 13), when deployed to node `N` and involving component `S` deployed to a distinct node `M` (lines 11–12), might have been caused by network congestion between `N` and `M` or by disconnection of node `N` (line 14). The Failure Enhancer suggests avoiding placing `C` (flavoured `FC`) onto node `N` by including a node avoidance constraint, viz., `avoid(d(C,FC),N)` (line 10). Indeed, the link between nodes `N` and `M` might be continuously subject to congestion or faulty. A symmetric clause (not shown) exists to handle the symmetric situation (i.e., congestion or disconnection of node `M`) by suggesting an `avoid(d(S,FS),M)` constraint.

The third clause of `suggested/1` identifies that component `C` (flavoured `FC`), deployed to node `N` (line 16), was unreachable or experiencing an internal error at time `T` (line 17). This failure overlapped with node `N` overloading of resource `R`, due to another component `S` (flavoured `FS`) racing with `C` for the resource `R` (line 18). The Failure Enhancer suggests avoiding placing `C` and `S` onto the same node through an *Anti-affinity* constraint between the two components in their current flavours, viz., `antiaffinity(d(C,FC),d(S,SF))` (line 15).

The fourth clause of `suggested/1` identifies a failure event (line 21) affecting component `C` when deployed to `N` in flavour `FC` (line 20). If the failure was due to node overloading (in the absence of a race) or disconnection (lines 22–25), the Failure Enhancer suggests avoiding placing `C` (flavoured `FC`) onto node `N` by including a node avoidance constraint, viz., `avoid(d(C,FC),N)` (line 19).

3.2 Energy enhancer

The Energy Enhancer generates soft deployment requirements within R that aim to reduce the environmental impact of the MSA execution[4]. Particularly, the Energy Enhancer is responsible for enabling carbon-efficient and environmentally conscious application deployments across the cloud continuum. This component ingests multiple inputs, including the *Application Description* (i.e., the services to be deployed), the *Infrastructure Description* (i.e., the available computing nodes), the current *Grid Carbon Intensity*, and relevant *Monitoring Metrics* that capture the runtime behavior of applications. Based on these inputs, the component produces a set of constraints that inform and guide the Solver during the generation of a suitable deployment plan. Moreover, the Energy Enhancer continuously improves its outputs by iteratively learning from past deployments, enabling the system to adapt to changes in both application behavior and infrastructure conditions.

The solution incorporates several key functionalities that work together to generate and refine environmentally aware deployment constraints. First, information about the carbon intensity of the underlying infrastructure is continuously gathered and processed. We assume that this information can be retrieved through external services, providing fine grained timely information about the electricity source composition in a specific geographical region.[5] Instead of relying on instantaneous values that might not be representative, the system considers aggregated measurements over a recent observation window (e.g., average behaviour since the last deployment cycle), resulting in more stable and representative data for decision-making.

In parallel, the carbon footprint associated with application services and their inter-service communications is estimated by analysing historical monitoring data. This allows the system to enrich the initial deployment descriptions with energy profiles that capture both computational and communication-related energy consumption.

Using these enriched inputs, the solution derives deployment constraints that reflect the current application behavior and infrastructure conditions. These constraints are defined according to pre-established templates and rules, which can be extended to accommodate new types of carbon-aware policies as needed. By doing so, the system remains flexible and adaptable to evolving sustainability objectives.

To ensure that the generated constraints build upon prior knowledge, previously learned information is retrieved, refined, and incrementally updated over time. This continuous learning process helps maintain consistency across multiple deployment cycles, avoiding the loss of useful insights from past configurations.

Given the potentially large number of constraints that may emerge, the solution applies ranking and filtering techniques to retain only those with the highest expected impact on energy efficiency and carbon emissions. Each constraint is assigned a normalized importance weight proportional to its estimated environmental impact, enabling relative comparison across constraints. Constraints with negligible absolute or relative impact are attenuated or discarded based on predefined thresholds. The threshold τ is set equal to a chosen quantile of that distribution. A sensitive analysis of the impact of the threshold value on the results is discussed in [23], showing that the relative importance of constraints decreases when lowering the threshold, confirming that the most impactful constraints are concentrated at higher quantile levels. This prioritization step ensures that the resulting set of constraints remains both relevant and manageable.

3.2.1 Knowledge base

To make deployment decisions more effective, the generation of constraints should rely on knowledge accumulated over time. For this purpose, the solution maintains a dedicated knowledge base that stores different types of information related to applications, their communications, the underlying infrastructure, and previously generated constraints.

The knowledge base keeps historical records of how each application service behaves in terms of energy consumption. Since a service may exhibit different energy profiles depending on where and how it is deployed, the system maintains information about typical ranges of its environmental footprint, including minimum, maximum, and average values observed from monitoring data collected during past deployments. In addition, the system stores information about the carbon impact of data exchanges between services. By analysing historical communication patterns, it builds a profile that reflects the environmental cost of interactions between different services across various deployments. The knowledge base also includes historical data on the carbon intensity of infrastructure nodes. Because the environmental characteristics of nodes can change over time, these records provide typical emission levels, which can be used to make better-informed placement decisions during future deployments.

Beyond raw monitoring data, the knowledge base preserves previously generated constraints. Each stored constraint contains information about its estimated environmental impact at the time of generation, along with a memory weight that reflects its current relevance. This memory weight decreases and finally decays if the constraint is not

[4] The pseudocode describing the constraints generation process is shown in Listing 1 (Appendix 11.).

[5] An example is https://www.electricitymaps.com

regenerated over multiple iterations, ensuring that outdated information gradually loses influence. The rate of decay, here expressed by ρ, is not fixed but should be set according to the stability of the infrastructure (e.g., how often nodes composition and environmental characteristics change).

The knowledge base is continuously enriched with newly collected data and recently generated constraints, while simultaneously updating the relevance of older ones. At each iteration, new deployment constraints are produced by observing the current data available, and valid past constraints are retrieved to complement them. This combined use of fresh observations and accumulated knowledge allows the system to generate more accurate, context-aware, and sustainable deployment decisions over time.

3.2.2 Enhancing environmental sustainability

The Energy Enhancer generates two types of constraints: *Affinity* and *Avoid*, with the same meaning of the constraints generated by the Failure Enhancer but with different conditions.

Avoid constraints are related to the energy consumption of individual services. Their goal is to limit deployments that would lead to high energy usage or emissions. When a service is known to consume excessive energy on a given node, a recommendation is generated to avoid that particular deployment. *Affinity* constraints target the energy overhead caused by service interactions. If two services exchange large amounts of data, deploying them on separate nodes may result in significant communication energy costs. In such cases, the system recommends co-locating the services to reduce this overhead.

The underlying logic for these two types of constraints can be expressed using Prolog clauses:

```
suggested(avoid(d(C,FC), N)) :-
    highConsumptionService(C, FC, N).

suggested(affinity(d(C,FC), d(S,FS))) :-
    dif(C, S),
    highConsumptionConnection(C, FC, S, FS).
```

The first clause produces recommendations to avoid placing certain service-flavour combinations on specific nodes when monitoring data indicates that such deployments are energy-inefficient. The generation of *Avoid* constraints relies on historical information saved in the knowledge base. For each possible combination of service, flavour, and node, the system evaluates whether deploying that specific configuration would result in excessive energy use or carbon emissions. This is determined by combining the service's energy profile with the node's carbon intensity and comparing the result against a predefined threshold. Whenever the estimated impact exceeds the threshold, the system generates an *Avoid* constraint to recommend avoiding the deployment of that service-flavour pair on the corresponding node. This ensures that environmentally costly placements are excluded from the deployment planning process.

```
antiaffinity(frontend,large,load_balancer,large).
#Constraint generated by the Failure Enhancer.
affinity(frontend,large,load_balancer,large).
#Constraint generated by the Energy Enhancer.
```

Fig. 3 Example of conflicting soft constraints generated by the Enhancers

The second rule generates recommendations to place two interacting services on the same node when their communication pattern would otherwise lead to a high carbon footprint. The generation of *Affinity* constraints is based on historical information saved in the knowledge base. For each potential combination of source service, its flavour, and destination service, the system evaluates whether their interaction is associated with high energy consumption. This assessment is performed by analyzing the energy profile of their communication and comparing it against a predefined threshold. If the estimated communication cost exceeds this value, the system generates an *Affinity* constraint. These constraints guide the scheduler to place the involved services on the same node. Together, these two constraints form the foundation for greener deployment.

3.3 Harmonizer

The Harmonizer component takes as input the enriched description of the application requirements `R+` (Fig. 1). Its primary function is to identify and resolve potential conflicts among the soft requirements, guided by the priorities defined by DevOps engineers, whether emphasizing resilience, sustainability, or balance between them. After processing, the Harmonizer produces `r+`, a refined subset (or, in some cases, the complete set) of requirements, which is subsequently forwarded to the Solver for consideration in the next deployment phase.

Figure 3 illustrates a case of conflicting soft constraints generated by the Failure Enhancer and the Energy Enhancer. In this example, the Failure Enhancer suggests deploying the *Frontend (Large flavour)* and *Load Balancer (Large flavour)* services on different nodes to improve fault tolerance through *antiaffinity*. Conversely, the Energy Enhancer proposes placing both services on the same node, suggesting an *affinity* constraint to minimize carbon emissions. When such conflicts occur, the Harmonizer resolves them according to the DevOps engineer's defined priorities: if failure is prioritized, the *affinity* constraint is discarded; if energy is prioritized, the *antiaffinity* constraint is removed. In the

absence of an explicit priority, both conflicting constraints are ignored in the subsequent deployment.

It is important to note that the Harmonizer does not possess a complete view of the MSA. As previously described, its role is limited to verifying that the soft constraints generated by the two enhancers are mutually consistent. If the generated soft constraints conflict with other infrastructure requirements, e.g., an avoid constraint applied to the only node capable of hosting a critical component, the Harmonizer is unable to detect or correct such conflicts directly. In these cases, if the Solver determines that the deployment with all constraints provided by the Harmonizer is unsatisfiable, the problem is re-executed multiple times. A detailed description of the Solver and how it works is provided in Sect. 3.4.

3.4 Solver

Following the description of the FREEDA framework [7] and its model (cf. Sect. 2), the application `A` and the infrastructure `I` are converted into a MiniZinc [24] data file. MiniZinc is a high-level, solver-independent modeling language for expressing constraint satisfaction and optimization problems. It enables users to define models in a standardized form that can be executed on multiple solvers without modification.

As described in Fig. 4, the configuration files `A` and `I` are updated based on the data collected during previous calls of the Enhancers (i.e., from the second call of the Solver onward, energy values for each component are revised, failed nodes are removed from the configuration, and resource usage and availability are updated accordingly). The new constraints generated by the analyzers after harmonization, together with the updated configuration files, are then re-converted to reflect the changes into a new MiniZinc model. This new model adopts the objective function described in Equation 1 to minimize component relocations between nodes, thereby improving deployment stability across successive calls of the solver.

The solver now possesses the full view of the MSA i.e., the requirements of each component, the capability of each node and the constraints that have already been harmonized, when present. As mentioned in Sect. 3.3, the constraints provided by the Harmonizer must be integrated with those defined by the FREEDA model [7]. If no additional constraints are present, the solver attempts to compute an optimal deployment according to the criteria defined by the FREEDA model [7], namely maximizing the importance of the flavours assigned to each deployed component. This is typically the case during the initial deployment, when no prior deployment data are available. If no feasible deployment can be found, the solver returns unsatisfiable and the toolchain terminates, explicitly informing the user that no valid deployment exists for the given application and infrastructure. Conversely, if a feasible deployment is identified, the solution is passed to the next stage of the toolchain as the best deployment plan `D`.

However, when additional constraints are provided by the Harmonizer, merging a priori cannot guarantee that the deployment will be satisfiable, as the interaction between constraints is not easily predictable in the general case. Formally, let H denote the set of soft constraints returned by the Harmonizer. FREEDA initially attempts to solve the deployment problem using the full set H, together with the hard constraints defined by the deployment model. In this initial attempt, the optimization objective is to minimize the number of component relocations between consecutive deployments, as specified in Eq.1.

If the resulting problem is satisfiable, the solver returns the best solution found within the allocated time. Otherwise, FREEDA progressively relaxes the soft constraints. The relaxation process starts by removing exactly one constraint from H, following the priority order produced by the Harmonizer, from the least important constraint to the most important one. For each relaxed configuration, FREEDA generates a new optimization problem and solves it independently using a one-minute timeout. As soon as a satisfiable configuration is found, the corresponding best solution is returned and the relaxation process terminates.

Instead, if no feasible deployment is found among all configurations obtained by removing a single constraint, FREEDA proceeds to configurations obtained by removing pairs of constraints, then triples, and so on. If no valid deployment can be identified after all possible relaxations

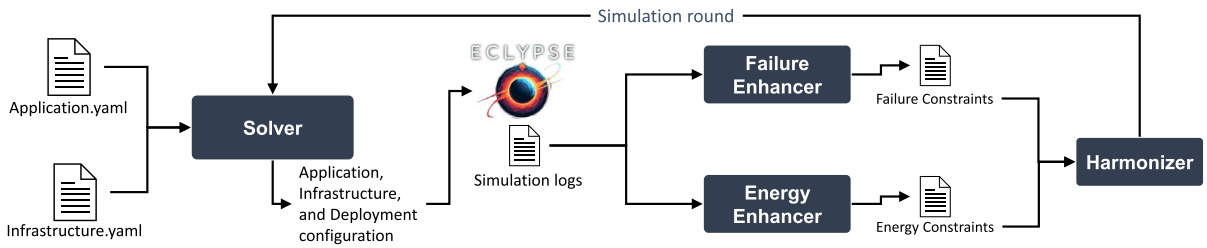

Fig. 4 Overview of the simulation toolchain

have been explored, the toolchain terminates and explicitly informs the user that no satisfactory deployment could be found.

This procedure guarantees that, whenever a feasible solution is found within the time limit, FREEDA returns a solution satisfying a subset of H of maximum cardinality. When multiple subsets of the same cardinality are feasible, ties are broken according to the Harmonizer priority order, thereby preserving the most important soft constraints whenever possible. The returned deployment is therefore compatible with all hard constraints and optimized, among the feasible relaxed configurations considered, according to the objective in Eq. 1.

4 Simulation

To assess whether the FREEDA toolchain enabled a failure-resilient, carbon-aware, and quality-aware deployment of MSAs over Cloud-Edge infrastructures, we run a simulation in which we compared different configurations of the FREEDA toolchain against a naïve *best-fit* placement policy, which places services on nodes satisfying their deployment requirements while also trying to maximize the utilization of infrastructure node resources without exceeding their capacity. This section provides an overview of the entire simulation toolchain, followed by detailed descriptions of each step in the subsequent subsections.

As shown in Fig. 4, the toolchain begins with the *YAML* [25] descriptions of the infrastructure and application, which are provided as input to the Solver. The information included in the YAML files is assumed to be readily available and can be automatically obtained from the underlying infrastructure and its runtime monitoring systems (e.g., Kurbenetes [21] natively supports getting the resources available in a node cluster). These files are parsed into a MiniZinc data file, which is passed to the Solver. The latter produces a deployment plan, which is then translated into an *ECLYPSE* [26] configuration.

ECLYPSE [26] is a publicly available[6] Python-based framework for simulating the definition, placement, deployment, and runtime behavior of multi-service applications (such as MSAs) over heterogeneous Cloud–Edge infrastructures. It relies on a formal environment model that supports fine-grained specification of both network topologies and multi-service applications, complemented by configurable tooling for common modeling tasks. We selected *ECLYPSE* because it natively supports the entire Cloud–Edge continuum, from resource-constrained edge devices to fog layers and cloud regions, thereby enabling the instantiation of heterogeneous and multi-layered infrastructures required for our evaluation.

ECLYPSE simulations are executed in multiple rounds, each combining an infrastructure scenario and an application scenario. In our case, the scenarios simulate failures or energy spikes within each simulation round. Application scenarios typically increase the workload or demand on specific components, while infrastructure scenarios reduce the available resources of certain nodes. This approach enables systematic and controlled experimentation across a wide range of operational conditions.

After each round, the simulation logs are processed by both the Failure Enhancer and the Energy Enhancer, which generate additional constraints and update the YAML files (both with new energy values and new availability of each infrastructure node) to improve the deployment. These constraints are then passed to the *Harmonizer*, which resolves immediate inconsistencies by prioritizing either failure resilience or energy efficiency, depending on the preference of the user. The harmonized constraints are subsequently fed into the Solver, which integrates them directly into the next deployment to adapt to changes in the infrastructure or application imposed by the scenarios, without ever discarding any of them. The process is iterative, with each new simulation round executed on the updated deployment.

4.1 Simulation setup

4.1.1 Application/infrastructure description

To run the simulation, we consider a reference architecture, abstracting a real-world MSA taken from a company (Fig. 5b). In the figure, nodes represent the application components, arrows indicate the dependencies of each component, and the available flavours of each component are shown above each node. All components are conceptualized as services that can be deployed on different machines. The application comprises several components and is representative of a large class of MSAs [27]. Specifically, the *Load balancer* distributes incoming client requests evenly across multiple instances of the frontend service. The *Frontend* interfaces with users, forwarding requests to backend services via the *API* service. In turn, the *API* service acts as the central business logic hub, processing requests from the *Frontend*, handling core application functionalities, and orchestrating calls to other backend components. Among these, *Redis* works as a caching layer, storing transient data for fast access, useful to accelerate response times and offload frequent database queries, while the *Database* manages persistent data storage required by the application. The last two components are the *Identity provider*, which handles user authentication and authorization, while *etcd* manages

[6] https://github.com/eclypse-org/eclypse

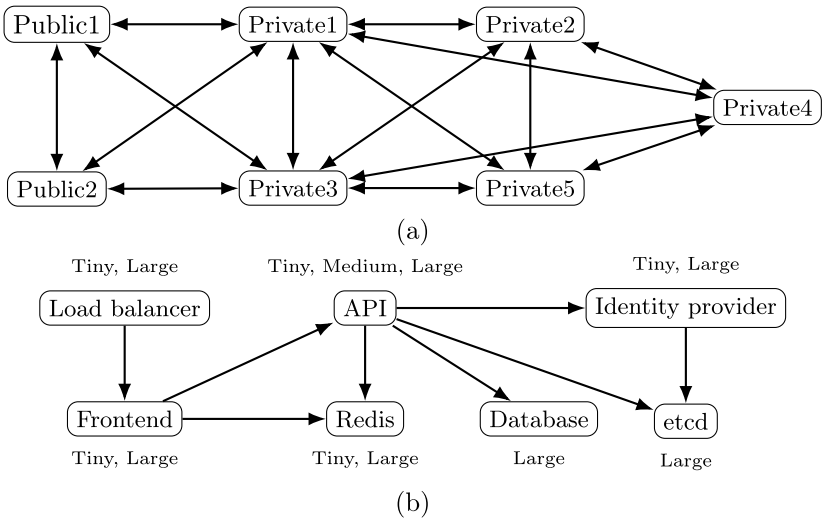

Fig. 5 Overview of the (**a**) Cloud-Edge infrastructure and (**b**) MSA considered in our simulation. The available flavours for a microservice are listed along such microservice

distributed configuration and service discovery. Flavour-wise, the Load Balancer, *Frontend*, *Redis*, and *Identity Provider* can be deployed in either *Tiny* or *Large* flavours; *etcd* and the *Database* are available only in the *Large* flavour; and the API component is offered in three flavours: *Tiny*, *Medium*, and *Large*. The application specification is defined through a YAML [25] file, following the format described in [7]. The full description of the application is available in a companion repository[7], defined using the FREEDA YAML specification [28].

We also define a representative infrastructure for the simulation. Specifically, we consider an Edge-Cloud infrastructure with a highly interconnected mesh topology with multiple redundant paths between nodes, providing resilience and load distribution capabilities [29]. The infrastructure is illustrated in Fig. 5a, where nodes represent the physical nodes (either public or private) where components can be deployed. The arrows in the figure denote physical connections between nodes, namely network links characterized by latency and availability attributes. When two components exhibit mutual dependencies, the model enforces their placement on physically connected nodes to ensure feasible communication. In the infrastructure, we find nodes *Public1* and *Public2*, which are connected Cloud nodes, enabling direct within-cloud communication. The *Private* nodes (5 in total) form a complete subnetwork of connected on-premises edge devices. Note that the only way for the Public nodes to directly communicate with services deployed in the on-premises Private subgroup is through nodes *Private1* and *Private2*.

Furthermore, each node has a *subnet* attribute so that only the services with a matching set of attributes can only be deployed on the corresponding nodes, e.g., each node has in the *subnet* attribute either the value *private* or *public* (depending on the type of node) and each component has this attribute too so it can be deployed only on certain nodes; the full description of the application, infrastructure, and attributes is publicly available on GitHub 7.

4.1.2 Solver

As illustrated in Fig. 4, the Solver receives as input the YAML specifications of the application and infrastructure – either with their initial values or as updated by the Failure Enhancer and Energy Enhancer after each simulation round – together with the constraints generated by the harmonizer. The YAML specification files go through a parsing phase to obtain a MiniZinc representation, as described in [7].

Following what is specified in Sect. 3.4, the solver (selected from those available within the MiniZinc distribution and defaulting to Gecode 6.3.0 also used in our experiments) aims to produce an optimal deployment whenever one exists within a one-minute timeout, according to the criteria that have been chosen based on the simulation round (i.e., maximizing the importance of each flavour in the first round of the simulation or relocating the least amount of components in subsequent rounds). Once the computation is complete, the resulting deployment is written to a text file that is subsequently parsed into a *ECLYPSE* simulation object.

4.1.3 Failure enhancer

This component takes as input the simulation logs generated from *ECLYPSE* and, as described in Sect. 3.1, generates constraints of type *Affinity*, *Anti-affinity* or *Avoid*. Figure 4 illustrates the workflow of the Failure Enhancer during the simulation. The process begins with the *Simulation.log* file (Fig. 6) generated by the *ECLYPSE* simulator, which is processed to generate the Prolog facts required by the Failure

[7] https://github.com/FREEDA-Project/case_study/.

```
17:20:33|ECLYPSE|Simulation - Event Start-0 fired.
17:20:33|ECLYPSE|Simulation - Event Tick-0 fired.
17:20:33|ECLYPSE|Simulation - Event Enact-0 fired.
17:20:33|ECLYPSE|PlacementManager -
    Placement of case_study on case_study
17:20:33|ECLYPSE|PlacementManager -
    {load_balancer_large -> public1 |
    api_large -> private1 |
    frontend_large -> public1 |
    redis_large -> private3 |
    identity_provider_large -> private3 |
    database_large -> private5 |
    etcd_large -> private1}
17:20:33|ECLYPSE|Simulation - Event Tick-1 fired.
17:20:33|ECLYPSE|Simulation - Event Enact-1 fired.
17:20:33|ECLYPSE|PlacementManager -
    Placement of case_study on case_study
17:20:33|ECLYPSE|PlacementManager -
    {load_balancer_large -> public1 |
    api_large -> private1 |
    frontend_large -> public1 |
    redis_large -> private3 |
    identity_provider_large -> private3 |
    database_large -> private5 |
    etcd_large -> private1}
...
```

Fig. 6 Extract of a *Simulation.log* file

```
deployedTo(api, large, private1).
deployedTo(database, large, private5).
deployedTo(etcd, large, private1).
unreachable(frontend, 31).
unreachable(frontend, 32).
overload(public1, cpu, 31, 98).
...
```

Fig. 7 Extract of a *deployment.pl* file

Enhancer. These facts describe both deployment and failure events, covering components as well as nodes, as depicted in Fig. 7. Based on this input, the Failure Enhancer generates soft constraints aimed at improving the resilience of the current MSA deployment.

4.1.4 Energy enhancer

The Energy Enhancer also processes *ECLYPSE* 's output, together with the application and infrastructure description, to generate soft constraints aimed at improving the energy consumption of the deployed MSA. The generated soft constraints can be of the type *Affinity* or *Avoid*, as described in Sect. 3. In the generated constraints, the constraint type is associated with the services it refers to and with a weight, denoting the relative importance of the constraint itself among the generated constraints.

4.1.5 Harmonizer

The Harmonizer takes as input the soft constraints generated by the enhancers (Fig. 4). Its role is to identify potential conflicts among these constraints and resolve them according to the user-defined priority, which may emphasize resilience, sustainability, or a balance between them. Once the constraints have been processed, they are sent to the Solver to be considered for the next deployment.

4.2 Scenarios

In this section, we describe the scenarios evaluated against both the application and the underlying infrastructure, outlining the expected behavior and the observed outcomes under different operating conditions. The objective is to systematically assess the individual and combined contributions of the tools composing the FREEDA toolchain, by comparing the results obtained with different tool configurations across the considered scenarios. Notably, all the considered scenarios were "deterministic", i.e., they introduced controlled perturbations at given times, to ensure repeatable conditions and enable reproducible reactions to such varying conditions.

A scenario is defined as a set of rules that modify the behaviour of elements during a simulation round. These rules may affect infrastructure components, such as CPU or RAM availability on a node, or application-level aspects, such as the energy consumption of a service. Rules can also be applied selectively to specific time intervals, referred to as simulation ticks, providing flexibility in the types of experiments that can be conducted. During the experimentation phase, two recurring patterns were employed:

- *Constant modification*, where a resource is adjusted by a fixed value over a defined range of ticks.
- *Sinusoidal modification*, where a resource varies following a sinusoidal function, ensuring that the starting and ending values align.

The first pattern was used to stress test thresholds at which a deployment might lack sufficient CPU or RAM to host a service, thereby inducing failures and downtime within the system. The second pattern was designed to simulate abnormal but transient behaviours, e.g., where no issues are evident at the start or end values, showcasing how the FREEDA approach is able to detect such instances and manage them accordingly.

The *ECLYPSE* simulator incorporates a mechanism referred to as Update Policy, which enables controlled environmental dynamism by programmatically modifying both infrastructure resource capacities and application-level

requirements across simulation rounds. Within the experimentation phase, this mechanism was employed to introduce predefined scenarios at specific points in the simulation timeline. Figure 8 presents an example of the configuration schema used for this purpose. In this structure, the position within the array specifies the simulation round at which a particular policy is applied, as well as the system component it targets, either the infrastructure or the application. This flexible approach enables not only the execution of multiple scenarios but also the exploration of various combinations of scenarios within a single simulation.

The *ECLYPSE* simulator also provides integrated deployment strategies for placing applications onto nodes. Among these, the *best-fit* strategy selects the node that maximizes resource utilization without exceeding capacity. Particularly, the best-fit strategy satisfies the resource requirements of the services to be implemented, while also heuristically attempting to maximize the utilization of infrastructure node resources without exceeding their capacity. The goal of the defined scenarios and simulations is to compare outcomes obtained using *ECLYPSE* 's best-fit strategy, providing a naïve baseline, against those achieved with the FREEDA approach.

4.3 Results

As showcased in Fig. 8, the scenarios applied during the simulation phase aimed to tackle a situation where certain nodes degraded their CPU and RAM resources through a *constant modification*, while some services degraded their energy performances through a *sinusoidal modification*. Results for all simulated scenarios are publicly available on GitHub[8].

During the simulation process, given the lack of alternative approaches (see Sect. 7), we evaluated five different

```
application_policies = [
    Example 1: [scenario] * X  # we run this
        scenario X times on the application side
    Example 2: [scenario1] * i + [scenario2] * j
        # we run scenario1 i times and then we run
        scenario2 j times on the application side.
]
infrastructure_policies = [
    Example 1: [scenario] * Y # we run this
        scenario Y times on the infrastructure side
    Example 2: [scenario3] * n + [scenario4] * m
        # we run scenario3 n times and then we run
        scenario4 m times on the infrastructure side.
]
```

Fig. 8 Code snippet showing the configuration schema used to run the scenarios

[8] https://github.com/FREEDA-Project/case_study/tree/loop/Simulation/Experiments

configurations of the toolchain to observe how these components behave under identical scenarios. These toolchain configurations are summarized below:

1. ***ECLYPSE best-fit***: The application is deployed using only the placement strategy *best-fit* provided by the simulator.
2. **Only Solver**: The application is deployed using the deployment configuration generated by the Solver, i.e., it only enforces the DevOps-specified constraints, without considering the Enhancers.
3. **Solver and Energy Enhancer**: The application is deployed using the Solver 's deployment configuration, enriched with constraints generated by the Energy Enhancer based on simulation logs.
4. **Solver and Failure Enhancer**: The application is deployed using the Solver 's deployment configuration, combined with constraints generated by the Failure Enhancer from the simulation logs.
5. **Full FREEDA**: The application is deployed using the complete FREEDA toolchain, i.e., the Solver 's deployment configuration together with the constraints generated by both the Energy Enhancer and Failure Enhancer.

The first configuration relies on the *ECLYPSE* 's *best-fit* strategy, deploying all components in their highest-performing flavour (i.e., *Large*). The goal of this configuration is to epitomize a naïve placement policy, which does not consider the notion of flavours coming along with the FREEDA toolchain, yet satisfying all services' deployment requirements. In this setting, all seven components were successfully deployed across all six simulation rounds. During the second deployment (corresponding to round 1), *ECLYPSE* reallocated certain components to maximise node's resource utilization, after which it maintained this configuration, resulting in stable energy usage. With respect to failures, since the scenario is repeated, the downtime plot exhibits a consistent pattern of failures across rounds (Fig. 9, row 1).

The second row of Fig. 9 illustrates the results achieved with the *only Solver* configuration. As in the previous case, the Solver attempts to deploy all components in their most powerful flavour. However, since the Solver does not maximize resource utilization nor consider energy optimization, the energy consumption is higher than in the first configuration (as more nodes get involved). However, because the deployed configuration remains unchanged across all simulation rounds, energy usage remains stable throughout. The failure patterns are identical to those observed previously, as the Solver alone (without the Failure Enhancer and Energy Enhancer) lacks awareness of the events occurring during the simulation rounds.

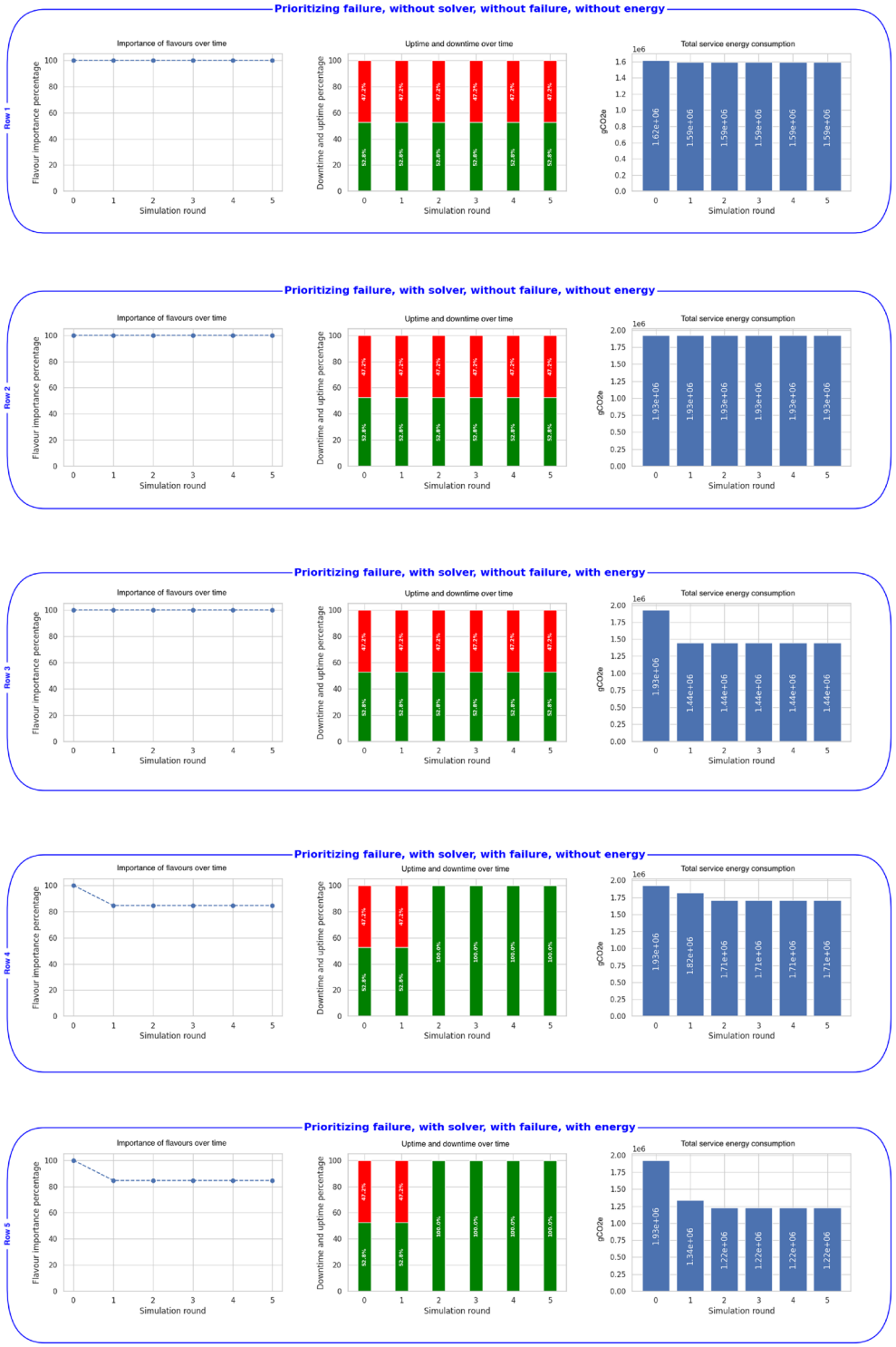

Fig. 9 Results obtained for our running example for each toolchain configuration, with priority failure. The initial placement round (i.e., round 0) used the objective function defined in [7], while the subsequent rounds used the objective function in Fig. 1

The Energy Enhancer is the module in charge of energy emissions from the deployment as a whole. The module is active in the third and fifth configurations of the simulation, whose results are shown in the third and fifth rows of Fig. 9. During the simulation it was put in place a scenario negatively impacting the *Database*, causing a spike in energy consumed from that service. Below in Fig. 10 we can see which constraints were generated from the Energy

```
avoid(d(identity_provider,large),private1,0.493).
avoid(d(identity_provider,large),private3,0.883).
avoid(d(identity_provider,large),private5,0.413).
avoid(d(database,large),private1,1.0).
```

Fig. 10 Soft constraints generated by the Energy Enhancer during the first simulation round

```
deployedTo(api, large, private1).
deployedTo(database, large, private5).
deployedTo(etcd, large, private1).
deployedTo(frontend, large, public1).
deployedTo(identity_provider, large, private3).
deployedTo(load_balancer, large, public1).
deployedTo(redis, large, private3).

unreachable(frontend, 31)...
...unreachable(frontend, 98).
unreachable(load_balancer, 31)...
...unreachable(load_balancer, 98).

overload(public1, cpu, 31, 98).
overload(public1, ram, 31, 98).
```

Fig. 11 Deployment.pl file generated by the Failure Enhancer during the first simulation round

```
avoid(d(frontend,large),public1).
avoid(d(load_balancer,large),public1).
```

Fig. 12 Soft constraints generated by the Failure Enhancer during the first simulation round

Enhancer after the first simulation round. The *Database* is correctly identified as the biggest energy consumer, and this is also reflected on the weights assigned, with the biggest priority being the *Database*. Since the *Database* requires an encrypted storage as a security measure for it to be deployed on a node, and being there only two nodes which such feature in the infrastructure used as an example, the Energy Enhancer will correctly propose soft constraints only for one of those two nodes, since we want to deploy the *Database* service somewhere.

For all the subsequent rounds the constraints generated from the Energy Enhancer remain the same, causing the energy consumption of the services to stabilise, as can be seen in Fig. 9.

The Failure Enhancer contribution is instead illustrated in the fourth and fifth rows of Fig. 9. When running, the constraints generated by the Failure Enhancer enable the Solver to maintain 100% uptime, even under resource degradation on the *Public1* and *Public2* nodes. It is important to notice that uptime is not achieved through service replication, but through reconfiguration of the application between simulation rounds. Figure 11 shows the input to the Failure Enhancer for the first simulation round, as generated by the *ECLYPSE* Parser from the simulation logs. In this round, both the *Frontend* and *Load Balacer* services were deployed on *Public1* using the *Large* flavour. Since the *best-fit* scenario deliberately degrades the resources of *Public1*, both services become unreachable between Ticks 31 and 98, as expected. Based on this information, the Failure Enhancer produces the soft constraints in Fig. 12, which are then passed to the Harmonizer and subsequently processed by the Solver.

4.4 Discussion

Observing Fig. 9, we can see how the full FREEDA approach is able to leverage all the improvements suggested by each single module and incorporate them in a single solution that is both failure resilient and energy observant at the same time.

We can also observe how the best-fit strategy, native to *ECLYPSE*, might achieve slightly better energy results than the configuration of FREEDA toolchain running only the Solver. However, we must recall that the *ECLYPSE* best fit does not account for the various deployment configurations needed, rather simply fitting the most consuming service in the greenest node, without properly checking if the service requires a private or a public node or the service dependencies, or other requirements, such as the encrypted storage for the *Database* service. This leads to the initial best fit deployment resulting greener than the FREEDA configurations without the Energy Enhancer component.

Additionally, from the failure-resilience standpoint, *ECLYPSE* 's best-fit strategy does not have ways to resolve problems that might arise, leading to continuous service disruptions. Conversely, the FREEDA toolchain in its full configuration achieves much better results (Fig. 9, row 5). At first, the energy consumption might result in higher values, due to the reasons explained previously, but they are swiftly brought down, below the initial deployment and even the *ECLYPSE* deployment strategy. This can be attributed to energy constraints being observed from the Solver and the it deciding to change the flavour of a component, thus diminishing the flavour importance too. From the second simulation round onwards, the situation stabilizes, with a successful maximization of flavour importance, while removing failures and minimizing energy consumption.

5 Emulation

This section provides an overview of the emulation toolchain, followed by detailed descriptions of each step in the subsequent subsections. The primary objective of the emulation phase is to gather data on the resilience and carbon efficiency provided by FREEDA. Specifically, we measure

service availability, application quality, and the carbon emissions produced by the cluster.

We executed all emulation experiments on a virtualized Kubernetes cluster deployed using Minikube, which enabled reproducible and controlled test scenarios. As in the simulation-based evaluation, we structured the emulation in discrete rounds. At the end of each round, we analyzed the observed events, including failures and requirement violations, and computed an updated deployment plan for the subsequent round. This experimental setup allows for the intentional introduction of performance degradation, adjustments in energy-consumption parameters to emulate carbon-intensity fluctuations, and imposes controlled resource constraints on selected infrastructure nodes.

5.1 Emulation setup

5.1.1 Reference application

To evaluate the FREEDA toolchain, described in Sect. 3, no existing benchmark was available that supported the management of multiple service flavours. Therefore, a new application called BrewMonitor was designed and implemented. BrewMonitor is an open-source, flavoured MSA, publicly available on GitHub[9] and composed of several interacting microservices that can be configured in two flavours (*Tiny* and *Large*). BrewMonitor emulates the collection, aggregation, and analysis of data within a realistic environment.

Fig. 13 illustrates the architecture of our reference application. BrewMonitor is a monitoring application for brewing plants, implemented as an MSA designed for modularity, scalability, and ease of extension. Its primary purpose is to collect real-time data from brewery sensors, aggregate and analyze it, and expose the results through REST interfaces for production operators and supervisory systems. The architecture was designed to support two distinct *flavours*: *Tiny* (red arrows in Figure 13) and *Large* (light blue arrows), which differ in functionality, data persistence, and resilience features. The MSA is composed of four main microservices, each responsible for a specific stage of the monitoring workflow:

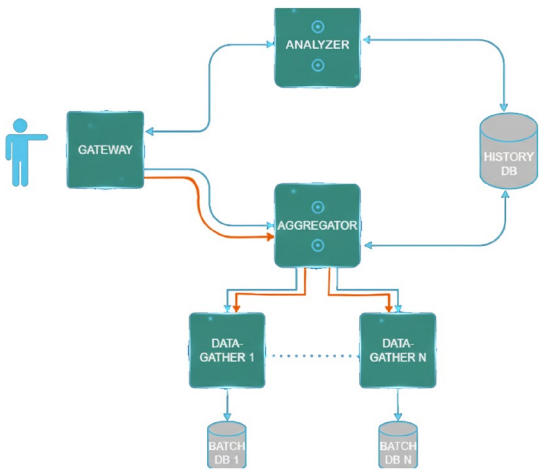

Fig. 13 Overview of the BrewMonitor MSA architecture

- The *Gateway* serves as the single entry point to the system, exposing HTTP REST endpoints that unify access to the underlying services. Implemented in Python using Flask, it provides lightweight integration, centralized logging, authentication, and rate limiting.
- The *Data Gathers* handle the sampling of process data (temperature, humidity, and pH) from simulated sensors. In the *Tiny* flavour, values are generated in memory upon request, providing a lightweight, stateless service. In the *Large* flavour, readings are persisted in MongoDB for 24 hours, enabling access to historical data for analysis and auditing.
- The *Aggregator* periodically collects data produced by one or more *Data Gather* instances. In the *Tiny* flavour, it queries the */data-gather/avg* endpoint and forwards the results directly to clients. In the *Large* flavour, a background thread polls each *Data Gather* instance every minute, stores the data in MongoDB, and exposes the latest aggregated records via the */aggregator/current* endpoint, ensuring consistency across collection nodes.
- The *Analyzer* is available only in *Large* flavour, this component performs statistical analyses on historical MongoDB data. It computes metrics such as maximum, minimum, standard deviation, and outliers for temperature, humidity, and pH, and exposes a single */analyzer/stats* endpoint that returns detailed JSON results for each monitored instance.

Cluster and node configuration. The experimental environment consists of a Minikube cluster composed of six virtual nodes, each with distinct technical specifications reflecting the heterogeneity typical of modern hybrid cloud environments:

- The *gatewaynodeefficient* node is publicly accessible and features 2 CPU cores, 4 GB of RAM, and a carbon intensity of 200 gCO2/kWh. This setup is ideal for hosting gateway services that prioritize stability over raw performance.
- The *gatewaynodestrong* node is a publicly accessible and features 4 CPU cores and 8 GB of RAM. It, however, exhibits a slightly higher carbon intensity of 250 gCO2/kWh.

[9] https://github.com/FREEDA-Project/brew-monitor.

- The *databasenode* node is equipped with 8 CPU cores and 8 GB of RAM and features a low carbon intensity of 100 gCO2/kWh.
- The *appnodestrong* node is equipped with 8 CPU cores, 8 GB of RAM, and optimized for a low carbon intensity of 100 gCO2/kWh.
- The *appnodeefficient* node features 6 CPU cores and 6 GB of RAM, but stands out as the most carbon-efficient node in the cluster, with a carbon intensity of only 90 gCO2/kWh. This node is particularly useful for testing the FREEDA toolchain's ability to prioritize carbon efficiency under favourable conditions.
- The *veryexpensive* node features 8 CPU cores and 8 GB of RAM, but is intentionally configured with a high carbon intensity of 1000 gCO2/kWh. It aims to showcase FREEDA's capability to avoid environmentally costly nodes, even if very powerful.

5.1.2 Metrics and monitoring

To monitor the reference MSA, a custom observation script was developed. This script adopts a multi-layered architecture for collecting and processing application metrics. By integrating native Kubernetes APIs, the Prometheus monitoring system, and Kepler (for energy consumption measurement), the script provides a comprehensive view of the cluster's performance at both the functional and energy levels.

During the emulation, three key metrics were monitored: *App Quality*, *Downtime*, and *C02 Emissions*. App Quality represents an aggregate measure of the importance of the flavours currently active within the MSA. It is expressed as the percentage ratio between the current importance score and the maximum theoretically achievable. This metric provides a quantitative indication of FREEDA's ability to maintain services running on the highest-performing flavours.

The Downtime Percentage quantifies the proportion of time during which the MSA is unable to deliver all required services at the minimum acceptable quality level. It is computed by continuously monitoring the readiness status of all application pods, excluding auxiliary pods (e.g., ballast pods used to simulate system stress). A period is considered uptime only when all expected pods are simultaneously in the Ready state, providing a realistic measure of overall service availability.

The total CO2 emissions constitute the primary metric for assessing the environmental impact of the MSA. They are calculated by integrating over time the product of instantaneous power consumption and the carbon intensity associated with each active node. Electrical power data are obtained via Prometheus queries, while carbon intensity values are retrieved from dynamic labels applied to the corresponding Kubernetes nodes.

5.2 Scenarios

The experimental suite defined for the emulation comprises seven scenarios across four thematic series, each targeting specific adaptive features of the FREEDA toolchain and its optimization mechanisms.

- **Series 1.x - Resource Exhaustion:** Tests FREEDA's handling of resource scarcity. Scenarios 1.1 and 1.2 artificially increase CPU demands for critical services, triggering rescheduling and activating failure-management logic.
- **Series 2.x - Nodal Stress and Carbon Variation:** Evaluates FREEDA's response to localized contention and environmental changes. Scenario 2.1 simulates heavy resource usage via ballast pods, while Scenario 2.2 dynamically raises node carbon intensity to test carbon-aware workload rebalancing.
- **Series 3.x - Failures and Policy Dilemmas:** Explores compound faults and trade-offs. Scenario 3.1 combines node failure with resource overload, and Scenario 3.2 introduces multiple stresses and conflicting optimization goals to assess FREEDA's decision-making under complex conditions.
- **Series 4.x - Carbon-Aware Adaptation:** Examines continuous adaptation to changing environments. Subscenarios 4.1 to 4.5 progressively vary carbon intensity, workload, and node capacity to test FREEDA's responsiveness to gradual and sudden shifts in both environmental and infrastructural parameters.

Notably, all the above-listed scenarios were designed and implemented to be "deterministic". Namely, each scenario introduced controlled perturbations at given times, with the goal of ensuring repeatable conditions and enabling reproducible reactions to such varying conditions.

This experimental suite provides a comprehensive assessment of FREEDA's resilience, efficiency, and environmental adaptability under diverse and evolving operational conditions. Here, the objective is to analyze what occurs when relying solely on native Kubernetes orchestration, without additional optimization logic, versus activating FREEDA to explicitly enforce failure resilience and carbon-efficiency objectives.

To this end, we execute each scenario sequentially under two distinct configurations. The baseline phase relies solely on native Kubernetes orchestration, with no intervention from FREEDA. The adaptive phase then activates FREEDA's optimization mechanisms to mitigate the effects

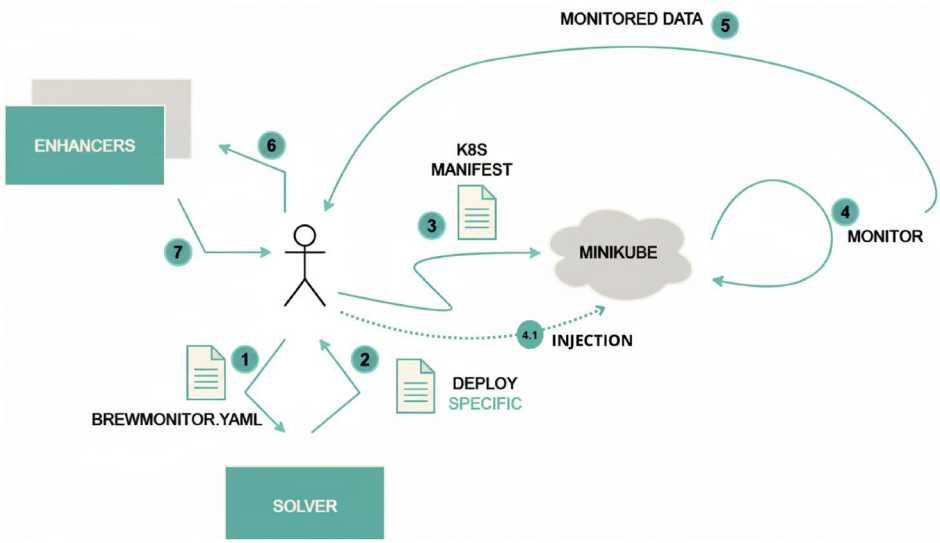

Fig. 14 Workflow followed during the execution of each emulated scenario

Table 1 Results of the seven emulated scenarios

	Kubernetes			FREEDA		
Reference Scenario	Average Flavour Importance	Average Uptime	Average Carbon Emissions	Average Flavour Importance	Average Uptime	Average Carbon Emissions
1.1	100%	100%	1.46e+02	25%	100%	4.38e+01
1.2	100%	56%	1.44e+02	62%	100%	5.35e+01
2.1	100%	46%	7.27e+01	62%	100%	5.6e+01
2.2	100%	100%	9.46e+01	25%	100%	4.79e+01
3.1	100%	49%	6.85e+01	82%	100%	5.18e+01
3.2	100%	39%	7.2e+01	76%	80%	5.61e+01
4.1	100%	60%	7.43e+01	64%	79%	5.7e+01

of the introduced degradations. This design enables a direct and quantitative comparison between the two configurations, isolating the influence of FREEDA's algorithmic decisions and allowing observed improvements to be confidently attributed to the framework's adaptive strategies.

Whereas the baseline scenarios rely on native Kubernetes orchestration, in contrast, the scenarios optimized with FREEDA implement a manual feedback loop to simulate an intelligent control system, as illustrated in Fig. 14. Each scenario is structured to first expose the limitations of traditional orchestration, documenting the failure patterns, energy inefficiencies, and performance bottlenecks that arise in the absence of intelligent optimization. The subsequent analysis focuses on the corrective actions implemented by FREEDA, e.g., service migration between nodes, flavour adaptation, or overall resource reallocation, demonstrating how these mechanisms enhance resilience, efficiency, and sustainability across the cluster.

5.3 Results

Table 1 summarizes the results derived from the seven emulated scenarios. To facilitate the reproducibility of our work, the information for all emulated scenarios, along with their execution instructions, are publicly available on GitHub.[10] The following discussion focuses specifically on the results from scenarios 2.2 and 3.1 to illustrate the performance of FREEDA.

During scenario 2.2, we implement a dynamic variation of the carbon intensity of the *appnodestrong* node, increasing it from 100 to 700 gCO2/kWh. This scenario evaluates FREEDA's carbon-aware capabilities, verifying if the framework is able to detect changes in environmental conditions and consequently rebalance workload placement to minimize the overall carbon impact. A detailed comparison between the baseline execution and the FREEDA-optimized deployment is illustrated in Figure 15. As shown, FREEDA's intervention fundamentally transforms the environmental profile of the MSA, demonstrating the potential of carbon-aware optimization strategies for modern MSAs.

The lower left panel of Fig. 15 shows that *App Quality* experiences an immediate, controlled reduction to approximately 25% from the first optimization round, remaining stable thereafter. This controlled reduction indicates that FREEDA implemented an aggressive but calibrated redeployment strategy. It involved a large-scale migration

[10] https://github.com/FREEDA-Project/brew-monitor/tree/sperimentazione

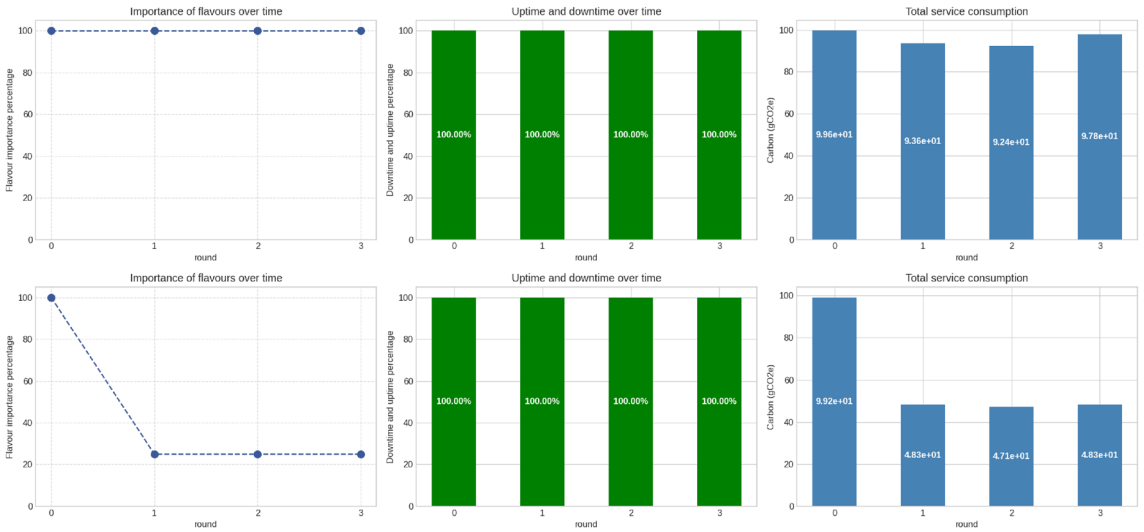

Fig. 15 Results of Scenario 2.2 with Kubernetes (first row) and FREEDA (second row). For FREEDA, the initial placement round (i.e., round 0) used the objective function defined in [7], while the subsequent rounds used the objective function in Eq. 1

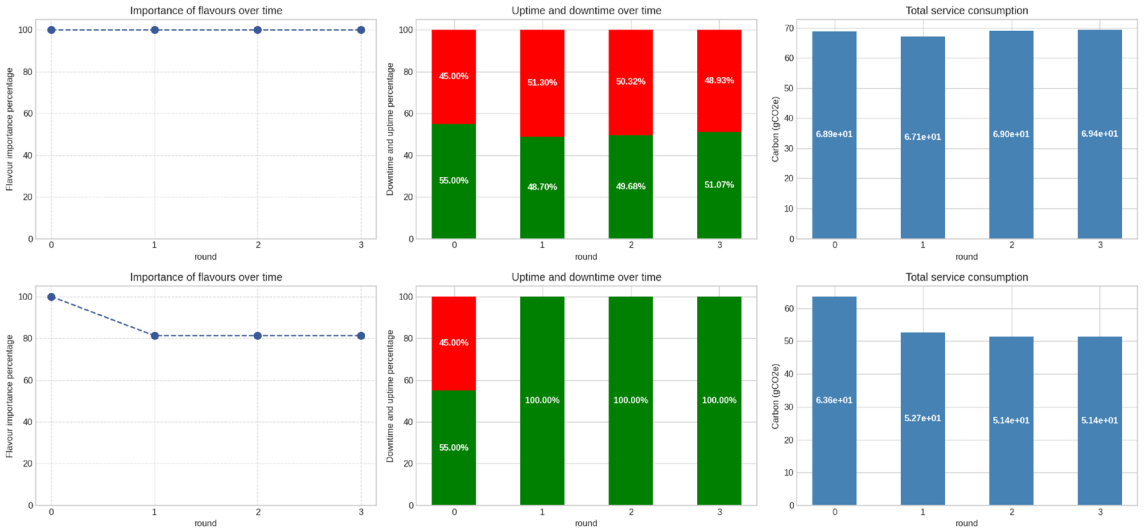

Fig. 16 Results of Scenario 3.1 with Kubernetes (first row) and FREEDA (second row). For FREEDA, the initial placement round (i.e., round 0) used the objective function defined in [7], while the subsequent rounds used the objective function in Eq. 1

of services from the high-carbon-intensity node to significantly more sustainable alternatives and the selective degradation of some flavours to optimize the overall MSA carbon efficiency. This strategy respects the trade-off between the carbon emissions budget and cost.

Throughout the experiment, *Uptime* (lower central panel) remains at 100%, which is enforced by the reconfiguration of the application between rounds. This demonstrates that FREEDA's carbon-aware optimizations do not compromise operational availability. Overall, the results confirm that FREEDA effectively balances sustainability and service continuity under dynamically changing environmental conditions.

Scenario 3.1 introduces a multiple-stress condition: first, it simulates the complete failure of the *gatewaynodestrong* node via drain, forcing the migration of all hosted pods; then, it subjects the *aggregator* service in its *Tiny* flavour to a resource overload. The goal is to test FREEDA's capacity to balance fault tolerance, performance, and environmental sustainability under critical stress conditions. A detailed comparison of the metrics between the baseline approach and the FREEDA-optimized solution for scenario 3.1 can be observed in Fig. 16.

The baseline execution of this scenario (Fig. 16, top row) highlights the severe impact of a node failure on an MSA not equipped with proactive resilience mechanisms. The sudden loss of an entire node represents a systemic shock,

exposing the limitations of native Kubernetes orchestration. As shown in the upper central panel of Fig. 16, the baseline MSA uptime fluctuates persistently between 45% and 55%. This indicates that the MSA spends almost half of the total time in a state of complete operational unavailability. This instability is not temporary but a permanent condition that persists for the entire experiment. This suggests that the native Kubernetes scheduler fails to stabilize the configuration after the node loss, specifically because there is no available node that can accommodate the *gateway* service with its *Large* flavour.

Consequently, the native orchestrator enters an infinite loop of failed scheduling attempts, repeatedly trying to reallocate orphaned services without ever converging to a valid configuration. This behaviour illustrates a fundamental limitation of standard orchestration in managing catastrophic failures.

By contrast, FREEDA's intervention demonstrates robust emergency management capabilities that overcome these structural limitations. Immediately after the perturbation, FREEDA stabilizes the MSA, restoring uptime to 100% and maintaining it throughout the experiment (see lower central panel of Fig. 16). Specifically, FREEDA automatically modifies the *gateway* flavour and migrates it to the only suitable node, *gatewaynodeefficient*, achieving both functional recovery and improved carbon efficiency.

During scenario 3.1, it was not possible to move services from the *appnodestrong* node to the *appnodeefficient* node, because the latter lacked sufficient resources to fully satisfy the demand. However, FREEDA still achieved an optimal trade-off between resilience and carbon optimization. This behaviour underscores FREEDA's design principle: maximize sustainability and resource efficiency without compromising the MSA quality or availability.

5.4 Discussion

The results presented in Table 1 demonstrate the effectiveness of the FREEDA approach across the seven emulated scenarios. In all cases, the FREEDA toolchain successfully eliminates downtime, transforming initially unstable deployments into configurations that achieve 100% availability. At the same time, FREEDA reduces CO2 emissions compared to the baseline configuration, with reductions ranging from 21% to 52%, with an average of 35% across all scenarios. This variability reflects the different optimization strategies adopted by FREEDA according to the specific characteristics of each scenario, including the nature and intensity of the induced stresses. The results highlight FREEDA's ability to manage complex scenarios, characterized by simultaneous multiple stresses and with gradual dynamic variations. Through reactive adaptations, the FREEDA toolchain adjusts the deployment configuration within the available operating margins, effectively mitigating the impact of infrastructure volatility on the running MSA. Even in the most critical case (scenario 3.2), where the baseline configuration exhibits an uptime between 34% and 43%, FREEDA achieves 100% availability while still achieving a CO2 emissions reduction between 21% and 24%.

6 Threats to validity

Based on the taxonomy developed by Wholin et al. [30], four potential threats to validity may apply to our study, namely threats to external, construct, internal, and conclusion validity. We considered these threats during the design and execution of the evaluation and adopted mitigation strategies to reduce their impact.

6.1 External validity

External validity concerns the applicability of a set of results to a more general context [30]. Our evaluation relies on a controlled experimental suite that combines simulation and emulation to represent heterogeneous Cloud–Edge infrastructures, dynamic carbon-intensity signals, and failure scenarios. However, we conducted the emulation on a Minikube-based environment, which cannot fully reproduce the physical distribution, scale, and hardware diversity of real Cloud–Edge deployments. In particular, node proximity effects and energy-efficiency implications of geographically distributed infrastructures cannot be fully captured when containers execute on a limited set of physical hosts. We mitigate this threat by complementing emulation with simulation-based experiments that explore larger infrastructures, heterogeneous configurations, and diverse workload profiles. We systematically vary environmental parameters, disturbance types, and deployment constraints to increase representativeness. Nevertheless, we acknowledge that generalization to very large-scale industrial environments remains limited. We thus plan to validate FREEDA on a realistic multi-node cluster and to further analyze its scalability, including a complexity assessment of the optimization process, to strengthen external validity.

6.2 Construct and internal validity

Wholin et al. [30] define the internal validity of studies as concerning the method employed to study and analyze data, including the potential types of bias involved. They instead define construct validity as the generalizability of the constructs under study. We addressed internal validity

by benchmarking FREEDA against established baselines such as Kubernetes orchestration and the *ECLYPSE best-fit* strategy. We also implement controlled stressors, including periodic resource fluctuations and abrupt node failures, to isolate the impact of our adaptation logic from external noise. Regarding construct validity, our work focuses on failure resilience, performance efficiency, and environmental sustainability. Each of these constructs exhibits multidimensional characteristics, and an incomplete operationalization could bias the interpretation of results. We mitigated this threat by grounding our metrics in established quality attributes for MSAs and sustainable computing research. We defined metrics such as application quality, downtime, and carbon emissions that directly reflect our main constructs. By employing multiple complementary metrics for each construct and aligning them with the optimization objectives encoded in FREEDA, we reduced the threat that a single representative measure would distort the conceptual interpretation. Additionally, to further reinforce the construct and internal validity of our findings, we released all sources needed to repeat our experiments on GitHub (see Sections 4 and 5).

6.3 Conclusion validity

Conclusion validity is defined as the degree to which the conclusions of a study are reasonably based on the available data [30]. We mitigated this threat by maintaining a rigorous experimental protocol and performing multiple iterations of each scenario to ensure consistency. We collected metrics via automated monitoring components and mocks, which were integrated into the experimental setup to reduce the probability of measurement errors. By systematically analyzing the collected data across multiple scenarios, we ensure that the reported improvements reflect consistent patterns rather than isolated outcomes. Moreover, we report both improvements and limitations, and we avoid extrapolating beyond the evaluated settings. These measures strengthen the credibility of our conclusions regarding FREEDA's capability to balance resilience, efficiency, and carbon awareness, while acknowledging the boundaries imposed by the current experimental scale.

7 Related work

Constraint reasoning was first applied to the optimal deployment of multi-service applications on cloud resources in [31–33]. Among these, [33] addresses service dependency modeling, whereas [31] and [32] focus on services' hardware, software, and availability requirements. More recently, constraint reasoning has been leveraged for generating containerized MSA deployments, with [34] and [35] extending the approach of [31] to microservice architectures, while [36] introduces container scheduling on Kubernetes based on QoS requirements. In contrast, [37] targets the deployment of MSAs on cloud virtual machines, encoding hardware and software requirements as constraints with the objective of minimizing overall deployment costs. In summary, existing approaches typically address isolated aspects of the MSA deployment problem, such as hardware/software requirements, service dependencies, or cost optimization, and assume a complete redeployment even when contextual changes affect only part of the deployment itself. Our proposal aims to bridge these gaps by providing a holistic MSA deployment over the Cloud-Edge continuum and continuous reasoning for adaptive MSA deployment over the Cloud-Edge continuum.

Similar considerations apply to the existing methods that exploit machine learning techniques to deploy multi-service applications over heterogeneous Cloud-Edge infrastructures. An overview of such methods is provided in a recent survey [38], which also classifies them based on whether featuring reliable resource provisioning, workload prediction, and adaptive placement decisions in highly dynamic Cloud-Edge infrastructures. Again, our proposal aims to complement these results by providing a holistic, adaptive MSA deployment over the Cloud-Edge continuum.

Existing approaches to enforcing failure resilience in MSAs mainly provide design and development guidelines [39, 40], data-driven self-healing and recovery mechanisms (e.g., MicroRAS [41]), or mechanisms for configuring deployment scripts to self-heal failing services, typically by restarting their hosting containers [42]. However, no existing solution supports the analysis of an already deployed MSA together with the available Cloud-Edge infrastructure, nor the automated enforcement of failure-resilient deployments over such infrastructures. Current techniques for failure analysis in MSAs primarily focus on detecting failures and identifying their root causes [43–46]. While these methods can automatically infer potential root causes for an observed failure, they generally stop at this stage, leaving DevOps engineers responsible for manually inspecting logs or monitored metrics to understand how the failure propagated throughout the system [47]. In summary, to the best of our knowledge, no existing technique currently enables the automated analysis of an MSA deployment over a Cloud-Edge infrastructure to enforce failure resilience within that deployment. Our proposal aims to fulfill this gap by providing an explainable enhancement of MSA deployments' failure resilience.

Various existing approaches focus on improving the energy efficiency of cloud data centers [48], while best practices for enhancing data center sustainability are outlined in

[49, 50]. However, computing infrastructures are increasingly distributed across the Cloud-Edge continuum, which is inherently composed of heterogeneous nodes. This heterogeneity causes significant fluctuations in power usage effectiveness [51], making the energy-aware allocation of Cloud-Edge resources an ongoing research challenge [52, 53]. Moreover, when applications are distributed over the Cloud-Edge continuum, their data must be stored and managed in close synergy with the applications themselves, both to reduce latency and to ensure compliance with security constraints. Despite this, current approaches to improving deployment efficiency generally focus only on the target infrastructure, occasionally extending to the temporal or geographical distribution of deployed applications [54–56]. Other works instead emphasize the application level [57, 58], supporting the design of energy-sustainable applications from scratch, with limited applicability to existing systems. More recently, energy-aware application decomposition and placement strategies have been proposed, particularly in serverless and edge contexts, where applications are decomposed into functions and placed across edge and cloud nodes to minimize energy consumption (e.g., FADE [59]). Meanwhile, the energy demand of deployed applications has grown alongside the widespread adoption of cloud computing, a trend expected to continue within Cloud-Edge infrastructures. This growing demand highlights the need for energy-aware MSA deployments across the Cloud-Edge continuum.

Overall, while prior work separately addresses energy-efficient infrastructures, application-level sustainability, or function-level placement, none integrates energy awareness with failure resilience and adaptive deployment for existing MSAs over heterogeneous Cloud-Edge infrastructures. Our proposal addresses this limitation by providing an explainable and adaptive deployment approach that jointly accounts for resilience, sustainability, and infrastructure heterogeneity.

8 Conclusions

We introduced a support for deploying flavoured MSAs over an heterogeneous Cloud–Edge infrastructure while jointly targeting their failure resilience and carbon efficiency. More precisely, we introduced the FREEDA toolchain, which automates the selection of microservice flavours and their placement across infrastructure nodes. By reasoning at the application level, FREEDA enables adaptive and multi-criteria deployment decisions that enforce failure resilience requirements and limit carbon emissions.

We also assessed FREEDA through controlled experiments simulating or emulating the realistic MSA deployments in Cloud–Edge scenarios. We considered scenarios including node failures, resource variability, and fluctuating carbon intensity, with the goal of assessing also FREEDA's ability to reconfigure MSA deployments. The results of our experiments show that FREEDA effectively adapts MSA deployments by suitably selecting/migrating microservice flavours, limiting unnecessary reconfigurations, and achieving a balanced trade-off between failure resilience and carbon efficiency.

For future work, we plan to engineer and extend FREEDA into a full-fledged support for deploying and managing MSAs across Cloud–Edge infrastructures. This includes (i) supporting deployment automation, e.g., by integrating FREEDA with Kubernetes, (ii) featuring advanced monitoring techniques for continuously assessing failure resilience and carbon-footprint of MSA deployments, also considering a broader set of failure- and carbon-specific metrics, and (iii) improving the overall deployment planning process by incorporating more refined strategies, e.g., bounded heuristic search or priority-based relaxation for resolving conflicting soft requirements. We also aim to (iv) enrich the FREEDA deployment model to natively support dynamic scaling and service replication, thus overcoming the current manual specification overheads.

Along the same line, we plan to extend FREEDA to support multi-tenant scenarios, namely to support deployinhg multiple MSAs over shared Cloud-Edge infrastructures. This requires extending the FREEDA deployment model to explicitly account for multiple MSAs, coordinating deployment decisions across such MSAs, and enhanced monitoring-data aggregation techniques. We then plan to validate the extended approach on realistic multi-node clusters.

Finally, we will investigate on how enhance FREEDA by integrating machine learning techniques with our constraint-based approach. Machine learning could help capture contextual information (e.g., temporal patterns and infrastructure dynamics), which is complex to express through logical abstractions, thus enhancing the generation/handling of soft constraints. Moreover, machine learning could be combined with our constraint solver to guide or accelerate the exploration of the solution space, enabling faster – yet effective – placement decisions.

A FREEDA model

For ease of reading, we now provide a summary of the FREEDA constraint model. The complete problem formulation of the model, along with a running example, is presented in [7].

Parameters

The input parameters are the constant values shaping a particular instance of the parametric model. Our model is parametric because the variables and the constraints are all expressed in terms of these input parameters. Table 2 summarizes the parameters that the FREEDA model uses in the backbone part. Full description of these parameters and their formulation is provided in [7].

Note that we are able to get the most powerful flavour of a component c thanks to the imp function; this function (synhesized by the user via multiple command-line parameters, as described in [7]) assigns an importance value to each flavour for each component, so that the model can always establish what is the best flavour for each component and maximize its sum.

Constraints

Constraints define the feasible solutions and encoding the deployment requirements. We enforce that each component c is deployed in at most one flavour, on at most one node:

$$\forall c \in \text{Comps} : \sum_{i \in \text{Flav}(c), j \in \text{Nodes}} D^c_{i,j} \leq 1$$

where $D^c_{i,j}$ is the binary indicator variable representing the deployment of the component c in flavour i on node j [7]. The constraint comes from the fact that flavours are different versions of the same component, and we want to impose the deployment of at most one version of the same component. At present, service-level replication requires duplicating the same service multiple times in the YAML description.

We also impose that each mandatory component m must be deployed by enforcing:

$$\forall m \in \text{MustComps} : \sum_{i \in \text{Flav}(c), j \in \text{Nodes}} D^m_{i,j} = 1$$

Additionally, we enforce that if component c is deployed in some flavour i, all the components it needs to run are deployed as well. Namely, for each pair $(c', i') \in \text{Uses}(c, i)$ component c' must be deployed in a flavour $k \in \text{Flav}(c)$ at least as powerful as i':

Table 2 Input parameters of FREEDA model. A full formalization, including examples and a more detailed explanation, can be seen in [7]. We use "Subset of$\mathbb{N}$" to indicate that these sets are non-empty and finite; w.l.o.g. they can therefore be identified with corresponding finite sets of positive integers.

Name	Domain	Description
Nodes	Subset of$\mathbb{N}$	Set of the nodes of the infrastructure
Comps	Subset of$\mathbb{N}$	Set of the components of the application
MustComps	Subset of Comps	Set of components that must be deployed
Flavours	Subset of$\mathbb{N}$	Set of all the flavours of all the components
Flav	$\text{Comps} \to \mathcal{P}(\text{Flavours}) \setminus \emptyset$	Set of all flavours of a specific component
imp	$\text{Comps} \times \text{Flavours} \to \mathbb{N}$	Denotes how important is deploying a component in a certain flavour
Uses	$\text{Comps} \times \text{Flavours} \to \mathcal{P}(\text{Comps} \times \text{Flavours})$	Expresses functional requirements between components
mayUse	$\text{Comps} \setminus \text{MustComps} \to \mathcal{P}(\text{Comps} \times \text{Flavours})$	Set of outgoing edges of a node in the dependency graph
CRes	Subset of$\mathbb{N}$	Set of consumable resources (e.g., CPU, RAM)
NRes	Subset of$\mathbb{N}$	Set of non-consumable resources (e.g., latency, availability)
Res	$\text{CRes} \cup \text{NRes}$	Set of all the resources
resReq	$\text{Comps} \times \text{Flavours} \to \mathcal{P}(\text{Res})$	Set of resources used by a component
resReq	$\text{Comps} \times \text{Flavours} \times \text{Comps} \to \mathcal{P}(\text{Res})$	Overload above function to denote the required resources for connecting two components
comReq	$\text{Comps} \times \text{Flavours} \times \text{Res} \to \mathbb{R}$	Minimum amount of resource required by a component in a certain flavour
depReq	$\text{Comps} \times \text{Flavours} \times \text{Comps} \times \text{Res} \to \mathbb{R}$	Minimum amount of resource required by the link connecting two components
nodeCap	$\text{Nodes} \times \text{Res} \to \mathbb{R}$	Maximum amount of resource available in a node
linkCap	$\text{Nodes} \times \text{Nodes} \times \text{Res} \to \mathbb{R}$	Maximum amount of resource available in a link
cost	$\text{Nodes} \times \text{Res} \to \mathbb{R}$	Per-unit cost of a resource on a node
carb	$\text{Nodes} \times \text{Res} \to \mathbb{R}$	Per-unit carbon emission of a resource on a node

$$\forall c \in \text{Comps}, \forall i \in \text{Flav}(c), \forall (c', i') \in \text{Uses}(c, i) :$$
$$\sum_{j \in \text{Nodes}} D^{c}_{i,j} \leq \sum_{\substack{k \in \text{Flav}(c') \, s.t. \, k \succeq_{c'} i', \\ j \in \text{Nodes}}} D^{c'}_{k,j}$$

Moreover, we forbid the deployment of isolated nodes by enforcing that, if a non-mandatory component c is deployed, then there must be at least one component c' that requires c deployed in some flavour i. Formally:

$$\forall c \in \text{Comps} \backslash \text{MustComps} :$$
$$\sum_{\substack{i \in \text{Flav}(c) \\ j \in \text{Nodes}}} D^{c}_{i,j} \leq \sum_{\substack{(c', i') \in \text{mayUse}(c) \\ j \in \text{Nodes}}} D^{c'}_{i',j}$$

We impose that if a component c deployed in flavour i requires a certain amount of resource r, then node_c must have capacity for r in $\text{nodeCap}(j, r)$. Furthermore, for consumable resources only, we must guarantee that a node fulfills the resource requirements for all the components deployed on it. We impose that that each node must have a sufficient quantity of a certain resource to meet the demands of all the components deployed on it.

To ensure the satisfaction of the dependency requirements $\text{depReq}(c, i, c', r)$ between two interdependent components c and c', we impose that each link between nodes must have support for the resources usage of all the couples of components on that link (i.e., $\text{depReq}(c, i, c', r) \geq \text{linkCap}(\text{node}_c, \text{node}_{c'}, r)$ for every couple of component deployed in node_c and $\text{node}_{c'}$). A full formalization can be seen in [7].

Finally, if β_{cost} is our money budget, and β_{carb} is our carbon budget, then we enforce $tot_{cost} \leq \beta_{\text{cost}}$ and $tot_{carb} \leq \beta_{\text{carb}}$.

Objective function. In the first deployment, FREEDA tries to deploy the best version of each component by maximizing the importance of each flavour [7], with such importance being statically defined by a function imp. This function assigns an importance value to each flavour for each component, so that the model can always establish what is the best flavour for each component and maximize its sum. In short, FREEDA tries to deploy the best (i.e., the most powerful) version of an MSA, given the above mentioned constraints. Thus, the objective function *maximises* the importance of deployed component flavours:

$$\sum_{\substack{c \in \text{Comps}, \\ i \in \text{Flav}(c)}} \text{imp}(c, i) \cdot \left(\sum_{j \in \text{Nodes}} \text{D}^{c}_{i,j} \right)$$

B Scalability assessment

We report on a comprehensive evaluation of the Solver performances, in order to show the sub-second resolution of a Solver run. In [7], we analysed the Solver scalability across various infrastructure topologies and solver configurations by running each test on Intel Core i5-4590 3.30GHz machines with 8 GB of RAM.

The objective of this analysis was to profile the performances of the Solver by considering different deployment problems of increasing size, by increasing independently nodes and components. To address this question, we randomly generated various different deployment configurations by tuning the number of components and nodes in the range [3..40]. We choose this range to reflect the scale of realistic applications.

We considered 3 different components' topologies commonly observed in microservice architectures: pipeline, small-world, and random. We also evaluate 5 infrastructure topologies: complete, small-world, random, ladder and wheel. Therefore, in total, we evaluated $(40 - 3 + 1)^2 \cdot 3 \cdot 5 = 21660$ different deployment scenarios.

For each component in every scenario, we randomly generated up to three different flavours. We fixed the number of resources to five and randomly determined, with uniform probability, whether a resource was consumable or not. To avoid trivially unfeasible deployments, we imposed that each individual component can be deployed on at least one node. The carbon and cost budget were set to enable each individual component to be deployable in its largest flavour, and we randomly determined whether it belonged to MustComps. We repeated this random generation 3 times to mitigate the side effects of randomness, hence we

Table 3 Performance with different solvers. For the complete analysis, please refer to [7]

solver	solved	error	avg all	avg solved	min	max	flat
OR-Tools	100.0%	0.0%	1.34s	1.34s	0.41s	56.56s	1.03s
Highs	99.07%	0.93%	N/A	1.39s	0.43s	300.0s	1.06s
Gecode	27.71%	0.0%	221.58s	20.23s	0.42s	300.0s	1.08s
Chuffed	20.0%	0.0%	242.73s	18.74s	0.43s	300.0s	1.16s

solved $21660 \cdot 3 = 64980$ instances run with 4 different solvers. Thus, we conducted a total of $64980 \cdot 4 = 259920$ experiments.

Because these problems are NP-hard, we set a timeout of 300 seconds. We considered that a solver solved a problem if it solved it to optimality or proved its unsatisfiability. If a solver could not solve a problem, its runtime was set to the timeout value, without any additional penalties.

Table 3 reports the results for each solver. All the runtimes include the flattening time from MiniZinc to FlatZinc. Column 'solved' shows the percentage of solved instances, while column 'errors' shows how many times, in percentage, a solver gave an incorrect answer (i.e., the solver reports unsatisfiability when the problem is satisfiable, or a wrong optimal value). Although limited to a small fraction of the dataset (0.93%), Highs was the only solver that provided unsound answers. OR-Tools instead solved all instances.

Unsatisfiable problems were around the 0.02% of all scenarios. Note that we cannot know a priori whether a problem we generated was satisfiable without running a solver. Furthermore, proving the optimality of a solution with incumbent objective value z^* actually involves proving the unsatisfiability of the same maximisation problem under the additional constraint $f_{\text{obj}} > z^*$, where f_{obj} is the objective function. Therefore, the bias towards satisfiable problems did not affect the generality and the validity of our experiments.

Column 'avg all' shows the average solving time over all the test instances. Because Highs provides some incorrect answers, we do not report its data. However, we can observe its performance in column 'avg solved', showing the average solving time over only those instances where the solver actually solved a problem (thus, excluding the incorrect answers for Highs and the problems not solved to optimality for Gecode and Chuffed). These results confirm the good performance of OR-Tools, also observable in columns 'min' and 'max', denoting respectively the minimum and maximum solving time for each instance: OR-Tools can solve every instance within 57 seconds. Notably, Chuffed flattening time, being a bit slower than Gecode, on average penalised its finding of the first solution.

Conversely, MIP solvers like Highs or hybrid solvers like OR-Tools, which incorporates MIP capabilities, are more effective for our case, likely due to the "quasi-linear" nature of the Solver model, having a linear objective function and predominantly linear constraints, hence the higher solving percentage. In addition, CP solvers such as Gecode and Chuffed were also considered, as their underlying technology differs significantly from MIP-based approaches. This choice reflects an intentional inclusion of diverse solving paradigms, where CP solvers provide strong constraint propagation and are well suited for combinatorial reasoning, complementing the strengths of MIP and hybrid methods.

Require: Application A, Infrastructure I, Monitoring M, Grid CI GCI, Knowledge Base KB
Ensure: Energy deployment constraints EC_{final}

1. Update Infrastructure Carbon Information
for all node $n \in I$ **do**
  retrieve recent carbon intensity for n from GCI
  update node profile in KB with this value
end for

2. Estimate Energy Profiles from Monitoring
for all components $c \in A$ **do**
  for all flavour fc of c **do**
    compute average computation energy of (c, fc) from M
    for all connected components s **do**
      compute average communication energy between (c, fc) and s
    end for
  end for
end for

3. Compute Carbon Footprint Values
$ImpactList \leftarrow \emptyset$
for all valid deployment option (c, fc, n) **do**
  estimate emissions of deploying (c, fc) on n
  add this value to $ImpactList$
end for
for all communication pair (c, fc, s, fs) **do**
  estimate emissions caused by communication
  add this value to $ImpactList$
end for

4. Determine Adaptive Threshold
compute the distribution of values in $ImpactList$
set emission threshold to the α-percentile of this distribution (e.g., $\alpha = 0.8$)

5. Generate Candidate Constraints
$C_{\text{new}} \leftarrow \emptyset$
for all deployment option (c, fc, n) **do**
  if estimated emissions exceed the service emission threshold **then**
    add AvoidNode$(d(c, fc), n)$ to C_{new}
  end if
end for
for all communication pair (c, fc, s) with $c \neq s$ **do**
  if communication emissions exceed the communication threshold **then**
    add Affinity$(d(c, fc), d(s, _))$ to C_{new}
  end if
end for

6. Update Knowledge Base
store updated energy profiles in KB
for newly observed constraints not in KB:
  insert into KB with impact Em_{new} and memory weight μ_0
for regenerated constraints already in KB:
  update impact value $Em \leftarrow (1 - \beta)\, Em + \beta\, Em_{\text{new}}$ and reset memory weight $\mu \leftarrow \mu_0$
for stored constraints not regenerated:
  apply memory decay $\mu \leftarrow \rho\mu$
remove constraints with $\mu < \mu_{\min}$
retrieve constraints with $\mu \geq \mu_{\min}$
$C_{\text{all}} \leftarrow C_{\text{new}} \cup$ valid stored constraints

7. Rank and Filter Constraints
normalize constraint importance based on estimated emission impact
discard constraints with negligible impact
$C_{\text{final}} \leftarrow$ remaining constraints
return C_{final}

Algorithm 1 GenerateEnergyConstraints

To get more insights on the impact of the topology on the solvers' performance, in Fig. 17, we plot the solving times of each solver on every test instance – sorted by increasing runtime – for all pairs of component-infrastructure topologies. As previously noted, Chuffed and Gecode solved fewer problems than Highs and OR-Tools. Overall, the results

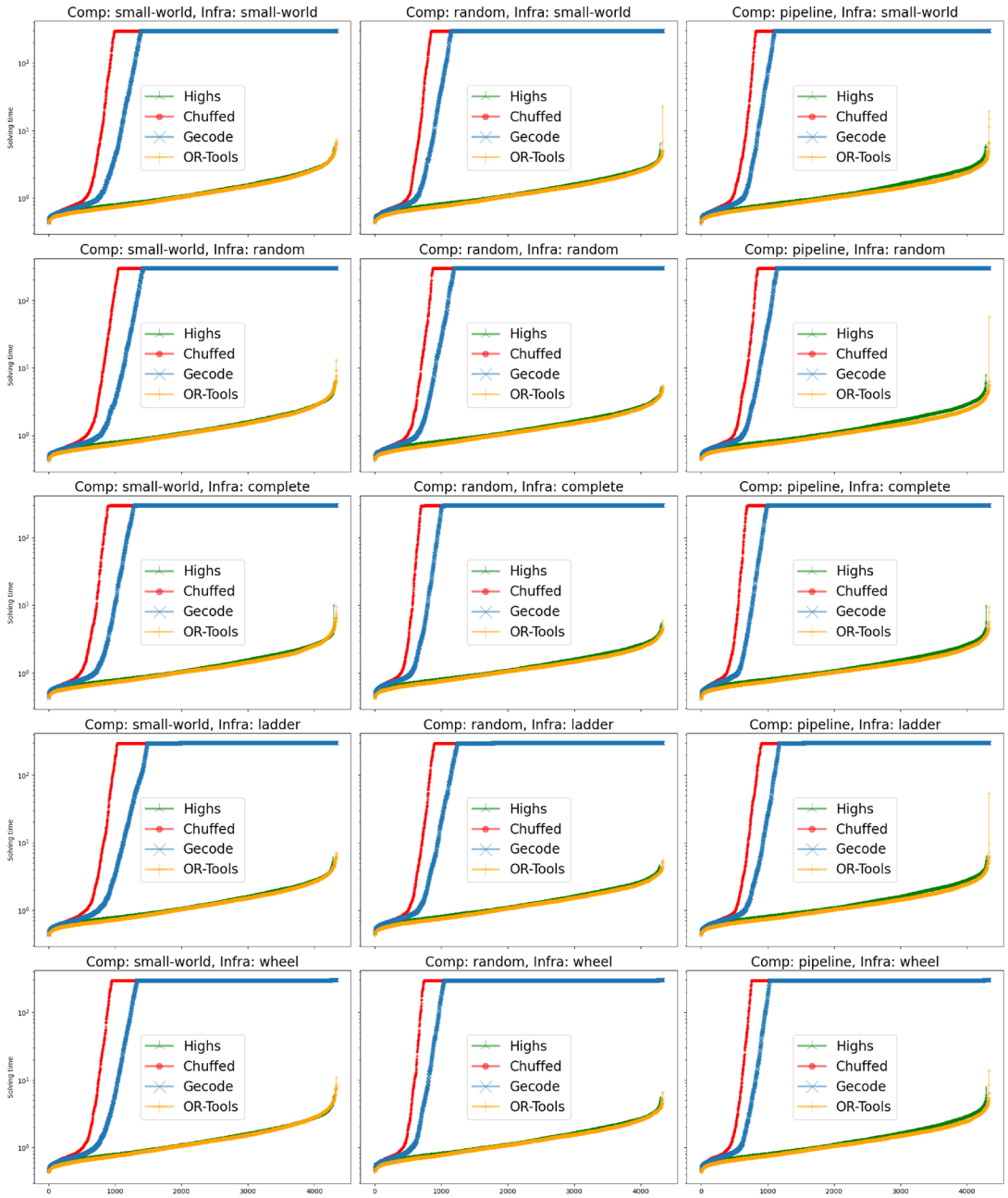

Fig. 17 Solving time in seconds (note the logarithmic scale) for each topology combination

suggest that the performance of the solvers is not strongly affected by variations in infrastructure or application topologies, as all plots in Fig. 17 exhibit a similar behaviour. The empirical results actually showed that Solver performs well when increasing the number of components and nodes, particularly when leveraging a solver based on mixed-integer linear programming. For a more in-depth analysis, please refer to [7].

C Energy enhancer: pseudocode

The Energy Enhancer is described in details in [23]. The pseudocode describing the constraints generation process is shown in Listing 1.

Acknowledgements This work was supported by the *FREEDA* project (CUP: I53D23003550006), funded by the frameworks PRIN (MUR, Italy) and Next Generation EU.

Author contributions All authors whose names appear on the submission made substantial contributions to the conception or design of the work; or the acquisition, analysis, or interpretation of data; or the creation of new software used in the work; drafted the work or revised it critically for important intellectual content; and approved the version to be published.

Data availability No datasets were generated or analysed during the current study.

References

1. Satyanarayanan, M.: The emergence of edge computing. Computer **50**(1), 30–39 (2017)
2. Ferrer, A.J.: Towards the decentralised cloud: Survey on approaches and challenges for mobile, ad hoc, and edge computing. ACM Comput. Surv. (2019). https://doi.org/10.1145/3243929
3. European Union: Strategic Agenda 2024-2029. https://www.consilium.europa.eu/en/european-council/strategic-agenda-2024-2029/ (2024)
4. European Union: Industrial Policy. https://www.consilium.europa.eu/en/policies/eu-industrial-policy/ (2022)
5. Park, G., et al.: Carbon-aware continuous learning for sustainable real-time machine learning analytics. In: Proceedings of the 21st European Conference on Computer Systems. EUROSYS '26, pp. 1640–1657. Association for Computing Machinery, New York, NY, USA (2026). https://doi.org/10.1145/3767295.3769361
6. Danushi, O., et al.: Carbon-efficient software design and development: A systematic literature review. ACM Comput. Surv. (2025). https://doi.org/10.1145/3728638
7. Gazza, S.: A constraint-based approach to optimise qos- and energy-aware cloud-edge application deployments. ACM Trans. Internet Technol. (2025). https://doi.org/10.1145/3757061
8. Bahreini, T.: Energy-aware capacity provisioning and resource allocation in edge computing systems. In: Zhang, T. (ed.) Edge Computing – EDGE 2019, pp. 31–45. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-23374-7_3
9. Toghani, M., et al.: ARGENT: Energy-aware scheduling in edge computing using energy valley optimizer. In: 2025 29th International Computer Conference, Computer Society of Iran (CSICC), pp. 1–6 (2025).https://doi.org/10.1109/CSICC65765.2025.10967461
10. Rahmati, M.: Energy-aware federated learning for secure edge computing in 5g-enabled iot networks. J. Elect. Syst. Inf. Technol. (2025). https://doi.org/10.1186/s43067-025-00203-2
11. Ahvar, E.: Deca: A dynamic energy cost and carbon emission-efficient application placement method for edge clouds. IEEE Access **9**, 70192–70213 (2021). https://doi.org/10.1109/ACCESS.2021.3075973
12. Huynh, D.V., et al.: Carbon-aware edge computing for internet of everything networks: A digital twin approach. IEEE Internet Things J. **12**(15), 29240–29251 (2025). https://doi.org/10.1109/JIOT.2025.3564157
13. Chen, X., et al.: From component to system: Rethinking edge computing design through a carbon-aware lens. SIGENERGY Energy Inform. Rev. **5**(2), 125–131 (2025). https://doi.org/10.1145/3757892.3757910
14. Zhang, G., et al.: Carbonedge: Carbon-aware deep learning inference framework for sustainable edge computing. CoRR **abs/2603.27420** (2026) https://doi.org/10.48550/ARXIV.2603.27420arxiv: 2603.27420
15. Abbasi-khazaei, T., Rezvani, M.H.: Energy-aware and carbon-efficient vm placement optimization in cloud datacenters using evolutionary computing methods. Soft. Comput. **26**(18), 9287–9322 (2022). https://doi.org/10.1007/s00500-022-07245-y
16. Tsenos, M., et al.: Energy efficient scheduling for serverless systems. In: 2023 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS), pp. 27–36 (2023). https://doi.org/10.1109/ACSOS58161.2023.00020
17. Stojkovic, J., et al.: Ecofaas: Rethinking the design of serverless environments for energy efficiency. In: 2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA), pp. 471–486 (2024). https://doi.org/10.1109/ISCA59077.2024.00042
18. Gaglianese, M., et al.: Green orchestration of cloud-edge applications: State of the art and open challenges. In: 2023 IEEE International Conference on Service-Oriented System Engineering (SOSE), pp. 250–261 (2023). https://doi.org/10.1109/SOSE58276.2023.00036
19. Forti, S., Brogi, A.: Declarative osmotic application placement. In: Polyvyanyy, A., Rinderle-Ma, S. (eds.) Advanced Information Systems Engineering Workshops, pp. 177–190. Springer, Cham (2021)
20. Vitali, M., et al.: Enriching cloud-native applications with sustainability features. In: 2023 IEEE International Conference on Cloud Engineering (IC2E), pp. 21–31 (2023). https://doi.org/10.1109/IC2E59103.2023.00011
21. The Kubernetes Authors: Kubernetes. https://kubernetes.io (2024)
22. Ponce, F., et al.: Enhancing failure resilience of cloud-edge microservices: The freeda approach. In: Pahl, C., Janes, A., Cerny, T., Lenarduzzi, V., Esposito, M. (eds.) Service-Oriented and Cloud Computing, pp. 105–111. Springer, Cham (2025)
23. D'Iapico, A., Vitali, M.: Green by Design: Constraint-Based Adaptive Deployment in the Cloud Continuum (2026). arxiv: 2602.18287
24. Nethercote, N., et al.: Minizinc: Towards a standard cp modelling language. In: Bessière, C. (ed.) Principles and Practice of Constraint Programming - CP 2007, pp. 529–543. Springer, Berlin, Heidelberg (2007)
25. YAML Language Development Team: YAML Ain't Markup Language (YAML™) Version 1.2.2. https://yaml.org/spec/1.2.2/. Accessed: 2025-01 (2021)
26. Massa, J., et al.: Eclypse: A python framework for simulation and emulation of the cloud-edge continuum. Journal of Software: Evolution and Process **38**(1), 70081 (2026). https://doi.org/10.1002/smr.70081
27. Francesco, P.D.: Architecting with microservices: A systematic mapping study. J. Syst. Softw. **150**, 77–97 (2019). https://doi.org/10.1016/J.JSS.2019.01.001
28. Simone Gazza, F.P., Soldani, J.: FREEDA YAML Model Specification (2024). https://github.com/FREEDA-Project/YAML-model/tree/v0.2
29. Sittón-Candanedo, I.: A review of edge computing reference architectures and a new global edge proposal. Future Gener. Comput. Syst **99**, 278–294 (2019). https://doi.org/10.1016/J.FUTURE.2019.04.016
30. Wholin, C.: Experimentation in software engineering: an introduction, vol. vol. 2, p. 274. Kluwer Academic Publishers (2000)
31. Ábrahám, E., et al.: Zephyrus2: On the fly deployment optimization using smt and cp technologies. In: Fränzle, M., Kapur, D., Zhan, N. (eds.) Dependable Software Engineering: Theories, Tools, and Applications, pp. 229–245. Springer, Cham (2016)
32. Di Cosmo, R., et al.: Automated synthesis and deployment of cloud applications. In: Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering. ASE '14, pp. 211–222. Association for Computing Machinery, New York, NY, USA (2014). https://doi.org/10.1145/2642937.2642980

33. Fischer, J., et al.: Engage: a deployment management system. SIGPLAN Not. **47**(6), 263–274 (2012). https://doi.org/10.1145/2345156.2254096
34. Bacchiani, L., et al.: Microservice dynamic architecture-level deployment orchestration. In: Damiani, F., Dardha, O. (eds.) Coordination Models and Languages, pp. 257–275. Springer, Cham (2021)
35. Bravetti, M., et al.: Optimal and automated deployment for microservices. In: Hähnle, R., Aalst, W. (eds.) Fundamental Approaches to Software Engineering, pp. 351–368. Springer, Cham (2019)
36. Lebesbye, T., et al.: Boreas - a service scheduler for optimal kubernetes deployment. In: Hacid, H., Kao, O., Mecella, M., Moha, N., Paik, H.-Y. (eds.) Service-Oriented Computing, pp. 221–237. Springer, Cham (2021)
37. Erașcu, M., et al.: Scalable optimal deployment in the cloud of component-based applications using optimization modulo theory, mathematical programming and symmetry breaking. J. Logic. Algebraic Method. Program. **121**, 100664 (2021). https://doi.org/10.1016/j.jlamp.2021.100664
38. Duc, T.L.: Machine learning methods for reliable resource provisioning in edge-cloud computing: A survey. ACM Comput. Surv. (2019). https://doi.org/10.1145/3341145
39. Giedrimas, V., et al.: The aspect of resilience in microservices-based software design. In: Mazzara, M., Ober, I., Salaün, G. (eds.) Software Technologies: Applications and Foundations, pp. 589–595. Springer, Cham (2018)
40. Heorhiadi, V., et al.: Gremlin: Systematic resilience testing of microservices. In: 2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS), pp. 57–66 (2016). https://doi.org/10.1109/ICDCS.2016.11
41. Wu, L., et al.: Microras: Automatic recovery in the absence of historical failure data for microservice systems. In: 2020 IEEE/ACM 13th International Conference on Utility and Cloud Computing (UCC), pp. 227–236 (2020). https://doi.org/10.1109/UCC48980.2020.00041
42. Brogi, A., et al.: Self-healing trans-cloud applications. Computing **104**(4), 809–833 (2022). https://doi.org/10.1007/s00607-021-00977-z
43. Aggarwal, P., et al.: Localization of operational faults in cloud applications by mining causal dependencies in logs using golden signals. In: Hacid, H., Outay, F., Paik, H.-Y., Alloum, A., Petrocchi, M., Bouadjenek, M.R., Beheshti, A., Liu, X., Maaradji, A. (eds.) Service-Oriented Computing - ICSOC 2020 Workshops, pp. 137–149. Springer, Cham (2021)
44. Liu, P., et al.: Unsupervised detection of microservice trace anomalies through service-level deep bayesian networks. In: 2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE), pp. 48–58 (2020). https://doi.org/10.1109/ISSRE5003.2020.00014
45. Meng, Y., et al.: Localizing failure root causes in a microservice through causality inference. In: 2020 IEEE/ACM 28th International Symposium on Quality of Service (IWQoS), pp. 1–10 (2020). https://doi.org/10.1109/IWQoS49365.2020.9213058
46. Wu, L.: MicroRCA: Root cause localization of performance issues in microservices. In: IEEE/IFIP Network Operations and Management Symposium (NOMS), (2020)
47. Soldani, J., Brogi, A.: Anomaly detection and failure root cause analysis in (micro) service-based cloud applications: A survey. ACM Comput. Surv. (2022). https://doi.org/10.1145/3501297
48. Gholipour, N., et al.: A novel energy-aware resource management technique using joint vm and container consolidation approach for green computing in cloud data centers. Simulati. Model. Practice Theory **104**, 102127 (2020). https://doi.org/10.1016/j.simpat.2020.102127
49. Acton, M., et al.: 2022 best practice guidelines for the eu code of conduct on data centre energy efficiency. European Commission Joint Research Centre (JRC), Brussles, Belgium, Tech. Rep. JRC128184 (2022)
50. Singh, H., et al.: Data center maturity model. Techn. Ber, The Green Grid (2011)
51. Jones, N.: How to stop data centres from gobbling up the world's electricity. Nature **561**(7722), 163–166 (2018)
52. Xiao, Y., Krunz, M.: Qoe and power efficiency tradeoff for fog computing networks with fog node cooperation. In: IEEE INFOCOM 2017 - IEEE Conference on Computer Communications, pp. 1–9 (2017). https://doi.org/10.1109/INFOCOM.2017.8057196
53. Zhang, W., et al.: Energy-efficient workload allocation and computation resource configuration in distributed cloud/edge computing systems with stochastic workloads. IEEE J. Sel. Areas Commun. **38**(6), 1118–1132 (2020). https://doi.org/10.1109/JSAC.2020.2986614
54. Nylander, T.: Brownoutcc: Cascaded control for bounding the response times of cloud applications. In: Annual American Control Conference (ACC), pp. 3354–3361. (2018). https://doi.org/10.23919/ACC.2018.8431282
55. Papadopoulos, A.V., et al.: Power-aware cloud brownout: Response time and power consumption control. In: 2017 IEEE 56th Annual Conference on Decision and Control (CDC), pp. 2686–2691 (2017). https://doi.org/10.1109/CDC.2017.8264049
56. Xu, M., et al.: A self-adaptive approach for managing applications and harnessing renewable energy for sustainable cloud computing. IEEE Transact. Sustain. Comput. **6**(4), 544–558 (2021). https://doi.org/10.1109/TSUSC.2020.3014943
57. Oliveira, F.G., Ledoux, T.: Self-optimisation of the energy footprint in service-oriented architectures. In: Proceedings of the 1st Workshop on Green Computing. GCM '10, pp. 4–9. Association for Computing Machinery, New York, NY, USA (2010). https://doi.org/10.1145/1925013.1925014
58. Nowak, A.: Automating green patterns to compensate CO2 emissions of cloud-based business processes. In: 8th International Conference on Advance Engineering Computing and Applications in Sciences, pp. 132–139. (2014)
59. Tzenetopoulos, A., et al.: Fade: Faas-inspired application decomposition and energy-aware function placement on the edge. In: Proceedings of the 24th International Workshop on Software and Compilers for Embedded Systems. SCOPES '21, pp. 7–10. Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3493229.3493306