A Constraint-based approach to Optimise QoS- and Energy-aware Cloud-Edge Application Deployments

SIMONE GAZZA, University of Bologna, Italy and OPTIMA ARC Centre, Australia
ROBERTO AMADINI, University of Bologna, Italy and OPTIMA ARC Centre, Australia
ANTONIO BROGI, University of Pisa, Italy
ANDREA D'IAPICO, Politecnico di Milano, Italy
STEFANO FORTI, University of Pisa, Italy
SAVERIO GIALLORENZO, University of Bologna, Italy and INRIA, France
PIERLUIGI PLEBANI, Politecnico di Milano, Italy
FRANCISCO PONCE, University of Pisa, Italy
JACOPO SOLDANI, University of Pisa, Italy
MONICA VITALI, Politecnico di Milano, Italy
GIANLUIGI ZAVATTARO, University of Bologna, Italy and INRIA, France

Cloud-Edge application deployment involves placing multiple software components on infrastructural topologies of heterogeneous nodes, ranging from Cloud servers to Internet-of-Things (IoT) edge devices. When multiple versions (or "*flavours*") of a component are available, application managers must select a flavour for each deployed component, and assign these components to specific nodes, all while considering constraints such as dependencies, quality of service (QoS), budget, operational costs, and carbon emissions. In complex scenarios, finding the optimal deployment is often infeasible for human operators without automated tools to systematically explore the solution space. To address this challenge, we introduce FREEDA, a first constraint optimisation approach for deploying constrained and multi-flavoured applications on Cloud-Edge infrastructure topologies. We demonstrate the practical feasibility of FREEDA through experiments on a variety of realistic Cloud-Edge infrastructural topologies and component architectures. Furthermore, we benchmark FREEDA against Zephyrus, a comparable tool employing the same underlying solving technology. Empirical results show that FREEDA achieves strong scalability across a broad spectrum of realistic configurations and consistently outperforms Zephyrus.

CCS Concepts: • **Computer systems organization → Cloud computing**; • **Theory of computation → Constraint and logic programming**.

Additional Key Words and Phrases: Cloud-edge Continuum, Carbon-awareness, Constraint Solving and Optimisation

Authors' Contact Information: Simone Gazza, simone.gazza@unibo.it, University of Bologna, Bologna, Italy;, OPTIMA ARC Centre, Melbourne, Australia; Roberto Amadini, roberto.amadini@unibo.it, University of Bologna, Bologna, Italy;, OPTIMA ARC Centre, Melbourne, Australia; Antonio Brogi, antonio.brogi@unipi.it, University of Pisa, Pisa, Italy; Andrea D'Iapico, andrea.diapico@polimi.it, Politecnico di Milano, Milan, Italy; Stefano Forti, stefano.forti@unipi.it, University of Pisa, Pisa, Italy; Saverio Giallorenzo, saverio.giallorenzo2@unibo.it, University of Bologna, Bologna, Italy;, INRIA, Sophia Antipolis, France; Pierluigi Plebani, pierluigi.plebani@polimi.it, Politecnico di Milano, Milan, Italy; Francisco Ponce, francisco.ponce@unipi.it, University of Pisa, Pisa, Italy; Jacopo Soldani, jacopo.soldani@unipi.it, University of Pisa, Pisa, Italy; Monica Vitali, monica.vitali@polimi.it, Politecnico di Milano, Milan, Italy; Gianluigi Zavattaro, gianluigi.zavattaro@unibo.it, University of Bologna, Bologna, Italy;, INRIA, Sophia Antipolis, France.

ACM 1557-6051/2025/7-ARTZ
https://doi.org/XXXXXXX.XXXXXXX

ACM Reference Format:
Simone Gazza, Roberto Amadini, Antonio Brogi, Andrea D'Iapico, Stefano Forti, Saverio Giallorenzo, Pierluigi Plebani, Francisco Ponce, Jacopo Soldani, Monica Vitali, and Gianluigi Zavattaro. 2025. A Constraint-based approach to Optimise QoS- and Energy-aware Cloud-Edge Application Deployments. *ACM Trans. Internet Technol.* X, Y, Article Z (July 2025), 29 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

With digital services becoming essential to daily life, we need flexible, scalable solutions that work effectively across diverse, geographically dispersed infrastructures. This need grows more urgent with the advancement of AI systems, real-time analytics, and user-focused applications that require distribution of computation to both sustain load of and provide low-latency access to these services. From the operations side, maintainers have to deploy these services on disparate computing resources made available where and when they are needed—e.g., centralised in powerful Cloud data centres as well as distributed to edge locations closer to users and data sources—to provide performance, adaptability, and sustainability.

The Internet of Things (IoT) ecosystem has further reshaped this computing landscape, requiring the effective and efficient management of multi-component applications over computing, storage, and networking resources along a seamless Cloud-Edge continuum [6]. Many of those applications (e.g., augmented reality, remote surgery, online gaming) impose stringent Quality of Service (QoS) requirements, e.g., low latency or high bandwidth between deployed distributed components, which, if unmet, can cause significant performance degradation.

In this context, manually maintaining deployments that simultaneously satisfy multifaceted requirements has become impracticable, which led to the proposal of approaches for the specification of deployment requirements that support the automated deployment and adaptation (e.g., via re-deployment) of a given service architecture. Extensive research (e.g., surveyed by Apat et al. [6], Costa et al. [19], Islam et al. [44]) highlights the prominence of tackling the suitable placement and orchestration of multi-component applications in Cloud-Edge settings. Such a task involves optimising various QoS aspects alongside hardware, software, and cost requirements to tame the complexity of available infrastructure in terms of size, geographic distribution, dynamicity, and resource heterogeneity. Moreover, with the rising energy consumption and carbon footprint of the Information and Communication Technology sector, there is a pressing need to consider these aspects at every stage of the software lifecycle to mitigate the environmental impact [33, 48].

Some approaches also target application placement with the goal of reducing energy consumption or carbon emissions (e.g., Abbasi-khazaei and Rezvani [1], Ahvar et al. [3], Aldossary and Alharbi [4], Forti and Brogi [30], Gnibga et al. [37], Yu et al. [63]). However, only a few proposals address the adaptation of applications and their placement to changing contextual settings, target objectives, or volume of incoming requests [44]. Forti and Brogi [29] explore deploying application components in different functionally equivalent *flavours* (i.e., versions) based on operator preferences and target operational costs, employing a greedy strategy to minimise the latter. However, their approach does not account for energy or sustainability constraints. In [59, 60], flavours are exploited in the context of microservice-based applications to adapt the application workflow and mitigate the environmental impact of the application. However, their approach does not consider the deployment in heterogenous distributed infrastructures.

Motivation. We argue that supporting flavours in Cloud-Edge deployments is fundamental for achieving high levels of automation and flexibility for operators. In fact, all flavours of the same component provide the same functionalities, but these functionalities may come in different versions according to the specific flavour. Therefore, each flavour typically requires different resources and may involve deploying different components. To illustrate this concept, we introduce the following running example.

Example 1.1. Consider a multi-tier application for video analytics including a backend for object detection, a database for video storage, and a web frontend that allows users to review and possibly amend misclassified objects. In this architecture, both the frontend and the backend come in two distinct flavours. The "*cloud*" flavour represents the most powerful incarnation of each component, e.g., the frontend has real-time video processing previews and the corresponding backend version uses resource-intensive, full-scale recognition models to ensure maximum accuracy. Notably, in this case, the backend must use the storage services offered by the database. Conversely, the "*edge*" flavour is optimised for resource-constrained environments. The edge-based frontend minimises resource consumption through local caching and compressed assets, while the corresponding backend uses lightweight models designed for limited hardware capabilities that do not require the database.

This flexibility allows the application to handle different requirements, such as user demands (e.g., privacy concerns) and infrastructural constraints. For instance, an "edge" deployment might involve only the frontend and backend, both deployed on the Edge, to avoid the transmission of private data to the Cloud. However, it is reasonable to assume that the most powerful flavours subsume the functionalities offered by the less powerful ones. In other terms, the "edge" version of the frontend can work also with the "cloud" version of the backend. Hence, the application could also support a hybrid Cloud-Edge deployment, where all the resources of the less powerful edge node are dedicated to the frontend, while the backend runs—along with its companion database—on powerful nodes in the Cloud.

Summarising, by leveraging appropriate component flavours, operators can optimise application performance while addressing specific operational goals and constraints.

Our contribution. In this article, we present FREEDA (Failure-Resilient, Energy-aware, and Explainable Deployment of microservice-based Applications over Cloud-IoT infrastructures), a preliminary approach to automate the optimal flavour selection and the feasible placement of selected components, taking into account general Cloud-Edge requirements—e.g., component dependencies, infrastructure topology, resource availability, cost and carbon budgets. To the best of our knowledge, FREEDA is the first system that enables constraint-based deployment while explicitly modelling flavour-based resource types. The inclusion of flavours is a key distinguishing feature of our approach. We believe that this abstraction reflects real-world deployment scenarios more accurately and allows for finer-grained optimisation, as it supports reasoning both about where to place each component and also on which variant best fits the available infrastructure and deployment constraints. We envision the stakeholders of FREEDA as individuals or entities responsible for maintaining Cloud-Edge applications, who have access to the infrastructure and aim to optimise component placement while meeting all deployment constraints. Practically, FREEDA is a framework comprising many elements, such as a language for specifying the architecture components and infrastructure nodes and a model for capturing the deployment problem. In the following, we use terms like "FREEDA specification" and "FREEDA model" to indicate these elements. When clear by the context, for brevity, we use the term "FREEDA" to identify the specific component discussed in a given section.

More precisely, in this article, we focus on the component flavour selection and placement problem, proposing a constraint optimisation model that jointly: (*a*) selects a flavour and allocates a node for each deployed component, (*b*) ensures all deployment requirements are met, e.g., carbon budget/energy consumption, and (*c*) prioritises deploying the "most powerful" flavours based on the application owner's preferences. The FREEDA model supports the automatic placement of components by providing a formal model to synthesise feasible deployments. To implement this approach, the FREEDA framework provides users with a syntax for describing the components, infrastructure, and requirements using the YAML language [62]. Given a specification, the FREEDA framework compiles it into a parametric constraint optimisation problem that models the deployment. The problem is then solved by a constraint solver to determine the optimal deployment configuration. To enable compatibility with various solvers, the constraint model is written in the solver-independent MiniZinc language [52].

Notably, traditional placement problems are a class of combinatorial optimisation problems that are known to be NP-complete [45] and, as such, notoriously difficult to solve efficiently, especially as the problem size grows. This computational hardness comes from the exponential number of possible allocations of components to infrastructure nodes, further complicated by inter-component dependencies and various deployment constraints. We demonstrate the practical feasibility of FREEDA by benchmarking the scalability of our implementation across a range of realistic Cloud-Edge infrastructural topologies and architectures—FREEDA is agnostic to the chosen topology and does not assume structural constraints, e.g., contrary to solutions that focus on Cloud-only deployments and require the completeness of the topology's graph. We then contextualize FREEDA's performance within the existing literature, specifically comparing it with Zephyrus [11], a state-of-the-art tool for optimal Cloud deployment that is most similar to FREEDA. Like FREEDA, Zephyrus supports modelling the available computing resources and software components in terms of their functional interdependencies and resource consumption. It also allows specification of the optimisation problem using MiniZinc. However, unlike FREEDA, Zephyrus assumes complete infrastructure topologies where every computation node can access all the others—an unrealistic assumption for the Cloud-Edge case, where non-complete topologies are the norm. We provide a detailed comparison of FREEDA and Zephyrus using realistic configurations that both tools can handle, showing that FREEDA consistently outperforms Zephyrus.

This article builds on and extends work from an earlier conference version [5]. Here, we introduce the FREEDA specification for expressing architectural and infrastructural configurations, as well as deployment requirements. We also revisit and refine the constraint model, by incorporating the concepts of source and target flavours. Furthermore, we extend the proof-of-concept evaluation of the conference version with a more comprehensive validation, including a comparison with Zephyrus [11].

Summarising, the contribution of this paper include:

- Introduction of a constraint-based optimisation model for joint component flavour selection and placement, capturing key Cloud-Edge deployment requirements such as component dependencies, resource constraints, cost, and energy budgets of both components and machine nodes.
- Design and implementation of the FREEDA framework, including a YAML-based specification language and a solver-independent encoding of the deployment model in MiniZinc, enabling compatibility with various solvers.
- Evaluation of the scalability and effectiveness of FREEDA through an extensive experimental setting on realistic Cloud-Edge scenarios, and compare its performance against Zephyrus, a state-of-the-art tool, showing consistent improvements.

Paper structure. We start by presenting, in Section 3, the FREEDA specification language allowing the user to define components, their deployment requirements, and the available computing infrastructure. Then, in Section 4, we introduce FREEDA's constraint model, used to compute optimal deployments. Section 5 provides further details on FREEDA implementation. In Section 6, we conduct a quantitative evaluation of the performance of FREEDA. We discuss the related literature in Section 2, and we draw our concluding remarks in Section 7.

2 Related Work

In the literature, we can find many proposals that marry constraint optimisation approaches with the deployment of software architectures, mainly concerning the Cloud case.

The earliest works apply constraint reasoning to the optimal deployment of multiservice applications on Cloud resources. For example, Fischer et al. [28] focus on managing component dependencies, while Ábrahám et al. [2] and Cosmo et al. [17] address hardware, software, and availability requirements. More recently, we find proposals that exploit constraint reasoning to generate containerised microservice architecture deployments. Notably, Bravetti et al. [11] and Bacchiani et al. [8] adapt the general approach proposed by Ábrahám et al. [2] to

microservice-based applications, while Lebesbye et al. [46] concentrate on scheduling Kubernetes containers based on QoS requirements. In this line of work, Eraşcu et al. [24] explores deploying microservice-based applications on Cloud virtual machines, encoding hardware and software requirements as constraints to minimise overall deployment costs.

Looking at more recent, Cloud-Edge applications, we find approaches for supporting optimal component deployment. For instance, Deng et al. [22] address the deployment of microservice-based applications in Mobile Edge Computing (MEC) environments using an ad-hoc iterative approach combined with a branch-and-bound strategy to solve the problem. Similarly, Peng et al. [53] focus on the joint optimisation of service deployment and request routing in MEC environments. Peng et al. [53] tackle the complexities of microservice interdependencies by formulating the problem as a delay minimisation task, using mixed integer linear programming (MILP) and queuing analysis. Another example is by Hu et al. [42], who model the problem with queuing network analysis and propose heuristic algorithms for the horizontal scaling of microservices. Hu et al. [42] also introduce a reinforcement learning approach to minimise user waiting times and resource consumption. David and Erascu [20] propose a similar approach to the one introduced by Bravetti et al. [11], but focuses specifically on symmetry-breaking strategies to reduce the search space and enhance solver efficiency. Massa et al. [49] combine declarative programming with mathematical optimisation to solve application placement problems in a data-aware manner. Herrera et al. [40] introduce a MILP-based system for determining QoS-optimal placements, by considering hardware and network characteristics such as latency and bandwidth—Herrera et al. [39] expand Herrera et al. [40]'s work to integrate software-defined network considerations, balancing response times and deployment costs.

Broadening our scope, we find work that use other techniques to help solve the deployment problem. For instance, Brogi et al. [12] introduce probabilistic placement strategies that consider context, QoS, and cost to determine placements of multiservice applications on Cloud-IoT infrastructures. Other examples are by Xia et al. [61], who propose an approach to reduce application response times, while Maio and Brandic [47] focus on offloading IoT tasks. Zhao et al. [64] use stochastic modelling of edge infrastructures and Monte Carlo simulations to validate placement strategies in worst-case scenarios. Faticanti et al. [27] present a greedy algorithm for partitioning application services between Cloud and Edge resources, based on their throughput requirements. Similarly, Forti et al. [31] address security and QoS-assurance in software-defined network settings, using probabilistic programming to improve placement and routing decisions. Genetic algorithms have also been employed to address optimisation problems related to resource placement. For instance, in Hosseini Shirvani and Ramzanpoor [41] the authors propose a multi-objective genetic-based algorithm to optimise virtual machine placement in Cloud datacenters while minimising power consumption, resource wastage, bandwidth usage and network delays. Similarly, in Farzai et al. [26] the authors formulate the placement of IoT application modules on fog infrastructure as a multi-objective optimisation problem and minimise power consumption and bandwidth wastage while also considering fault tolerance thresholds to ensure reliable application execution.

All these works differ from our contribution because they do not focus on deploying multi-flavoured Cloud-Edge applications on generic infrastructure topologies, composed of heterogeneous nodes, from Cloud servers to IoT edge devices. Indeed, FREEDA stands out as the first constraint optimisation approach specifically designed to handle multi-flavoured applications under complex constraints, including dependencies, QoS, budget, operational costs, and carbon emissions.

3 FREEDA Specifications

In this section, we outline the three specification schemas provided by FREEDA [56] to define a deployment problem:

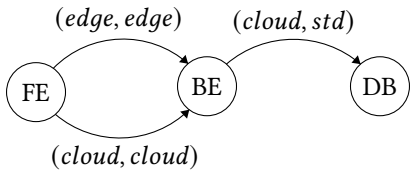


Fig. 1. Theorem 1.1's dependency graph.

- *application specifications* (Section 3.1), specifying the components, their flavours, and the dependencies between components in specific flavours;
- *deployment requirements* (Section 3.2), expressing both functional and non-functional requirements of the application, such as resource needs, quality of service, and budget constraints;
- *infrastructure specifications* (Section 3.3), describing the available computing infrastructure, including node capabilities, interconnection links, and their associated costs.

The language we use to implement these schemas is YAML (version 1.2) [62]. YAML is a human-readable data serialisation standard commonly used for configuration files and data exchange between languages with different data structures. This language emphasises simplicity and supports hierarchical data representation and is adopted by many Cloud tools such as Docker [50] and Kubernetes [13].

3.1 Application Specifications

In FREEDA, an application is represented by a *dependency (multi-) graph*, which is a labelled topological graph where nodes denote application components, and there is an edge labelled (s, t) from component c to c' if the deployment of c in the *source flavour* s requires the deployment of c' in a *target flavour* which is "*at least as powerful as* t". Indeed, since different flavours represent different versions of a component, we can safely sort them from its least powerful to its most powerful version.

As an example, Fig. 1 shows the dependency graph for Theorem 1.1, consisting of three nodes: frontend (FE), backend (BE), and database (DB). FE and BE have two flavours, i.e., "edge" (least powerful) and "cloud" (most powerful), whereas DB comes only in one "standard" flavour. If deployed in flavour cloud, FE requires BE to be deployed in flavour cloud, which in turn requires BE to be deployed in its standard flavour. Instead, when deploying FE in flavour edge, BE in flavour edge would suffice. Node BE has no outgoing edge with source flavour labelled "edge" because a backend in that flavour does not require any additional component.

Note that deploying the frontend in flavour edge does not necessarily imply deploying the backend in the same flavour. A backend in flavour cloud, which consequently triggers the deployment of the database, is acceptable since the cloud flavour is more powerful than edge. Clearly, the feasibility of such a deployment is subject to several constraints—e.g., there may simply not be enough budget available to deploy all these components.

The YAML encoding of Fig. 1 is shown in Listing 1. The key `name` identifies the application name, followed by the description of its `components`. The optional key `must` specifies whether the application needs the component to work, i.e., we cannot deploy the application without deploying this component. This feature enables the application owner to specify a set of *'entry-point'* components that must be deployed, e.g., to allow clients to interact with the application.

Each element of the key `flavours` contains the name of the source flavour, the `component`(s) that must be deployed if the source flavour is selected (specified by the key `uses`), together with the corresponding target flavour (key `min_flavour`) of these component(s). The target flavour is optional: if absent, the least important (namely, the "least powerful"), flavour of the target component is considered by default. The importance among

```yaml
name: video_analytics
components:
  frontend:
    must: true
    flavours:
      cloud:
        uses:
          - component: backend
            min_flavour: cloud
      edge:
        uses:
          - component: backend
            min_flavour: edge
    importance_order: [edge, cloud]
  backend:
    flavours:
      cloud:
        uses:
          - component: database
            min_flavour: standard
      edge:
        uses: []
    importance_order: [edge, cloud]
  database:
    flavours:
      standard:
        uses: []
    importance_order: [standard]
```

Listing 1. Application specification for Theorem 1.1.

the flavours of a component is defined by its `importance_order`, which defines a total order over the flavours of the component, from least to most powerful.

3.2 Deployment Requirements

The deployment requirements for an application can be of two types:

- *functional*: such as `cpu`, `ram`, `storage`, `bwIn` and `bwOut` (the latter two being the inbound and outbound bandwidth, respectively), with numerical values indicating the minimum amount required;
- *non-functional*: such as `availability` and `latency`, with numerical values indicating respectively the minimum and maximum amount required, and `security`, whose value is a list of the required security features, e.g., `firewall`, `ssl`, etc.

In the YAML encoding, we can specify both the functional requirements `common` to all flavours of a component, and those that are `flavour-specific`. For example, the specification in Listing 2 imposes that—regardless of the selected flavour—a deployed frontend must guarantee *at least* 10 Mbps of inbound and outbound bandwidth, 90% availability, and 8 GB storage. On the other hand, requirements such as CPU, RAM, and security levels depend on the chosen flavour. For example, if the frontend is deployed in the cloud flavour, it requires at least 4 GB of RAM, whereas in the edge flavour the minimum requirement is 2 GB.

Listing 2 also shows the `dependencies` requirements that we can express in YAML. These dependencies are specified as follows: if component c uses component c' in some flavour, then the infrastructural link connecting c and c' must fulfil certain requirements. For example, in Listing 2, we specify that if component `frontend` is deployed in flavour `edge`, then the link connecting the `frontend` to the `backend` must have *at most* a `latency` of 10 ms, and *at least* a 90% availability (key `avail`). Instead, if the `cloud` flavour is selected, the availability must be greater (98%) whereas the `latency` requirement is relaxed (20 ms).

FREEDA also allows users to set "budgets" for their applications, namely upper bounds on the application's monetary cost and the carbon footprint they are willing to incur, the latter expressed in gCO2-eq/KWh. These are represented by the `cost` and `carbon` keys in Listing 2.

```yaml
requirements:
  components:
    frontend:
      common:
        bwIn: 10
        bwOut: 10
        availability: 90
        storage: 8
      flavour-specific:
        edge:
          cpu: 1
          ram: 2
          security: [ssl, firewall]
        cloud:
          cpu: 2
          ram: 4
          security: [ssl, firewall, enc_storage]
    backend:
      common:
        bwIn: 10
        bwOut: 10
        availability: 90
        storage: 8
      flavour-specific:
        edge:
          cpu: 1
          ram: 2
          security:  [ssl, enc_storage]
        cloud:
          cpu: 2
          ram: 4
          security: [ssl, firewall, enc_storage]
    database:
      common:
        bwIn: 20
        bwOut: 10
        availability: 99
        cpu: 1
        ram: 8
        storage: 256
        security: [ssl, enc_storage]


dependencies:
  frontend:
    edge:
      backend: {latency: 10, avail: 90}
    cloud:
      backend: {latency: 20, avail: 98}
  backend:
    cloud:
      database: {latency: 20, avail: 99}


budget:
  cost: 600
  carbon: 500
```

Listing 2. Example of deployment requirements for Theorem 1.1 (`avail` is a shortcut for `availability`).

3.3 Infrastructure Specification

A deployment is a mapping of component flavours to infrastructure nodes, respecting a number of constraints on:

- *node resources*: the total amount of resources consumed by the components placed on a node must not exceed the resources available on that node. These resources, referred to as *consumable*, include CPU, RAM, storage, and bandwidth;
- *quality of service for components*: the QoS of the node hosting a component must satisfy the specified constraints. In particular: (*a*) the availability of a node must be at least the one required by the components it hosts, (*b*) the security of a node must include all security properties required by the components it hosts.

```yaml
nodes:
  n1:
    capabilities:
      cpu: 4
      ram: 8
      storage: 256
      bwIn: 100
      bwOut: 200
      availability: 90
      security: [ssl, firewall, enc_storage]
    profile:
      cost: {cpu: 50, ram: 5, storage: 1}
      carbon: 27
  n2:
    capabilities:
      cpu: 4
      ram: 8
      storage: 256
      bwIn: 100
      bwOut: 200
      availability: 95
      security: [ssl, enc_storage]
    profile:
      cost: {cpu: 50, ram: 5, storage: 1}
      carbon: 35
  n3:
    capabilities:
      cpu: 16
      ram: 32
      storage: 512
      bwIn: 500
      bwOut: 500
      availability: 99
      security: [ssl, firewall, enc_storage]
    profile:
      cost: {cpu: 100, ram: 10, storage: 1}
      carbon: 25


links:
  - connected_nodes: [n1, n2]
    capabilities: {latency: 10, avail: 98}
  - connected_nodes: [n2, n3]
    capabilities: {latency: 20, avail: 99}
```

Listing 3. Example of infrastructure specification for Theorem 1.1 (`avail` is a shortcut for `availability`).

Properties such as availability and security, which are not consumed during operation, are classified as *non-consumable* resources.

- *quality of service for dependencies*: the QoS of the link connecting two nodes must satisfy the requirements of the dependencies between the components hosted on those nodes. Specifically, if component c uses component c' in some flavour, then c can be placed on a node n and c' on a node n' only if: (a) the availability of the link $\{n, n'\}$ is *at least* that required by the dependency requirement between c and c', and (b) the latency of $\{n, n'\}$ is *at most* that specified by the dependency requirement between c and c';
- *budget constraints*: the total monetary and carbon costs associated with the consumable resources deployed—which may vary across nodes—must not exceed the specified budget.

The YAML specifications used to define infrastructural constraints are illustrated in Listing 3, where the infrastructure consists of three nodes: `n1`, `n2` and `n3`. For instance, `n1` and `n2` might represent edge nodes, while `n3` might be a cloud node. The infrastructure graph is assumed to be undirected and not necessarily complete; for instance, Listing 3 shows no link between `n1` and `n3`. This fact implies that any pair of components with a dependency constraint on their connection (e.g., the `frontend` and `backend` in Listing 2) cannot be deployed respectively in `n1` and `n3`. Therefore, these components must be assigned to other nodes or even to the same node, provided the node's available resources satisfy their requirements.

For example, the `frontend` cannot be deployed on node `n2`, because this node does not provide a `firewall`. Similarly, the `frontend` in flavour `edge` and the corresponding `backend` cannot be deployed on nodes `n2` and `n3`, as the latency of the link connecting these nodes (i.e., 20 ms) exceeds the maximum latency specified by their dependency requirements (i.e., 10 ms, see Listing 2).

4 Constraint Model

The specifications outlined in Section 3 serve as the foundation of the FREEDA toolchain, defining the key requirements for the desired deployment. However, multiple deployments—or in some cases, none—may fit these specifications. FREEDA automates the transformation of these specifications into an actual deployment by converting the YAML code into a constraint optimisation problem, i.e., an *abstract* mathematical model whose optimal solution corresponds to an optimal deployment w.r.t. the given specifications.

In this section, using Theorem 1.1 as a reference for examples, we formalise the main ingredients characterising the constraint model derived from the FREEDA specifications, namely:

- *parameters*, representing input data;
- *variables*, representing deployment decisions;
- *constraints*, representing deployment restrictions;
- *objective function*, representing what constitutes an optimal deployment.

4.1 Parameters

The input parameters are the *constant* values shaping a particular *instance* of the parametric model. Our model is parametric because the variables, constraints, and objective function are all expressed in terms of these input parameters. We formalise them as follows.

4.1.1 Components and Flavours. Let Comps be the set of components of our application and Flavours the set of all components' flavours. Because these sets are non-empty and finite, without loss of generality from now on we shall identify them with corresponding finite sets of positive integers $\{1, 2, 3, \dots\}$.

Let MustComps $\subseteq$ Comps be the components that *must* always be deployed. This is an important parameter defining the '*entry-point*' components that we must deploy to allow clients to interact with the application.

Let $\mathit{Flav} : \mathsf{Comps} \to \mathcal{P}(\mathsf{Flavours}) \setminus \emptyset$ be the function returning the non-empty set $\mathit{Flav}(c)$ of all flavours offered by component c. A *total* order $\preceq_c \subseteq \mathit{Flav}(c) \times \mathit{Flav}(c)$ is provided over the flavours of c, where $f \prec_c f'$ means that flavour f' is more "powerful" than f.

To enable cross-comparisons among flavours of different components and to weight the quality of different flavours, we also formalise the concept of *importance* with a function $\mathit{imp} : \mathsf{Comps} \times \mathsf{Flavours} \to \mathbb{N}$ such that $\mathit{imp}(c, f)$ denotes how important is deploying c in flavour f, the higher value of *imp*, the better. We assume that $f \preceq_c f'$ if and only if $\mathit{imp}(c, f) \leq \mathit{imp}(c, f')$, i.e., if flavour f' is more powerful than f, then it is also more important—the *imp* mapping can be synthesised from the importance order, as explained in Section 4.4.

The function $\mathit{Uses} : \mathsf{Comps} \times \mathsf{Flavours} \to \mathcal{P}(\mathsf{Comps} \times \mathsf{Flavours})$ returns the set $\mathit{Uses}(c, f)$ of all pairs (c', f') such that if c is deployed in flavour f, then c' *must* be deployed in a flavour g' *at least* as powerful as f', i.e., $f' \preceq_{c'} g'$. Therefore, the dependency graph introduced in Section 3 has Comps as set of nodes, and an edge

$$c \xrightarrow{(f,f')} c'$$

for each pair (c, f) such that $(c', f') \in \mathit{Uses}(c, f)$. Because the graph is topological, it must be $c \neq c'$.

We impose that a component not belonging to MustComps is only deployed if it is the target of an *active dependency*, i.e., we do not allow the deployment of components $c \in \mathsf{Comps} \setminus \mathsf{MustComps}$ not used by any other deployed component. To formalise this constraint, we use a function that associates to each component

the other components, and the corresponding flavours, that could use it. Namely, we define $mayUse : \mathsf{Comps} \setminus \mathsf{MustComps} \to \mathcal{P}(\mathsf{Comps} \times \mathsf{Flavours})$ as the function:

$$mayUse(c) = \{(c', f') \mid \exists f \in Flav(c).\ (c, f) \in Uses(c', f')\}$$

returning the set of all pairs (c', f') such that c' in flavour f' requires the deployment of c in some flavour. That is, each $(c', f') \in mayUse(c)$ denotes an edge in the dependency graph from source node c' to target node c.

For instance, in Theorem 1.1, we have $\mathsf{Comps} = \{\mathrm{FE}, \mathrm{BE}, \mathrm{DB}\}$ and $\mathsf{Flavours} = \{\mathrm{C}, \mathrm{E}, \mathrm{S}\}$ where C corresponds to flavour "cloud", E to "edge", and S to "standard". The flavours of each component are $Flav(\mathrm{FE}) = Flav(\mathrm{BE}) = \{\mathrm{C}, \mathrm{E}\}$ and $Flav(\mathrm{DB}) = \{\mathrm{S}\}$, with $\mathrm{E} \preceq_c \mathrm{C}$ for $c \in \{\mathrm{FE}, \mathrm{BE}\}$. The dependency graph in Fig. 1 is defined by $Uses(\mathrm{FE}, \mathrm{E}) = \{(\mathrm{BE}, \mathrm{E})\}$, $Uses(\mathrm{FE}, \mathrm{C}) = \{(\mathrm{BE}, \mathrm{C})\}$, $Uses(\mathrm{BE}, \mathrm{C}) = \{(\mathrm{DB}, \mathrm{S})\}$ and $Uses(\mathrm{BE}, \mathrm{E}) = Uses(\mathrm{DB}, \mathrm{S}) = \emptyset$. Reasonably, we expect the frontend to be the entry-point of the application (as represented in Listing 1), i.e., $\mathsf{MustComps} = \{\mathrm{FE}\}$. In this case, we would have $mayUse(\mathrm{BE}) = \{(\mathrm{FE}, \mathrm{C}), (\mathrm{FE}, \mathrm{E})\}$ and $mayUse(\mathrm{DB}) = \{(\mathrm{BE}, \mathrm{C})\}$.

4.1.2 Resources. We denote with $\mathsf{Res} = \mathsf{CRes} \cup \mathsf{NRes}$ the finite set of *resources*, defined by the disjoint union between its *consumable* (CRes) and *non-consumable* (NRes) resources, as defined in Section 3.3.

The auxiliary function $resReq : \mathsf{Comps} \times \mathsf{Flavours} \to \mathcal{P}(\mathsf{Res})$ returns the resources $resReq(c, f)$ required by c when deployed in flavour f. We overload this definition with $resReq : \mathsf{Comps} \times \mathsf{Flavours} \times \mathsf{Comps} \to \mathcal{P}(\mathsf{Res})$, denoting the set $resReq(c, f, c')$ of resources required for "connecting" the source component c, deployed in flavour f, to the target component $c' \in Uses(c, f)$.

A *component requirement* is a function $comReq : \mathsf{Comps} \times \mathsf{Flavours} \times \mathsf{Res} \to \mathbb{R}$ such that $comReq(c, f, r)$ is the *minimum* amount of resource $r \in resReq(c, f)$ required by c to be executed. A *dependency requirement* is instead a function $depReq : \mathsf{Comps} \times \mathsf{Flavours} \times \mathsf{Comps} \times \mathsf{Res} \to \mathbb{R}$ such that $depReq(c, f, c', r)$ is the *minimum* amount of resource $r \in resReq(c, f, c')$ required by the link connecting the "source" component c in flavour f to the "target" component c'.

Note that both *comReq* and *depReq* define *lower* bounds (e.g., *at least* 4GB of RAM needed). However, we may also require *upper* bounds (e.g., *at most* some latency time). For simplicity, and w.l.o.g., in the following we only consider lower bounds, as the case for upper bounds is symmetrical.

For instance, in Theorem 1.1, the required units of consumable resource CPU vary across the different components and flavours. For example, $comReq(\mathrm{FE}, \mathrm{E}, \mathrm{CPU}) = comReq(\mathrm{BE}, \mathrm{E}, \mathrm{CPU}) = comReq(\mathrm{DB}, \mathrm{S}, \mathrm{CPU}) = 1$ while $comReq(\mathrm{FE}, \mathrm{C}, \mathrm{CPU}) = comReq(\mathrm{BE}, \mathrm{C}, \mathrm{CPU}) = 2$. The availability required by a link connecting two components also differ: $depReq(\mathrm{FE}, \mathrm{E}, \mathrm{BE}, \mathrm{avail}) = 90$, $depReq(\mathrm{FE}, \mathrm{C}, \mathrm{BE}, \mathrm{avail}) = 98$, and $depReq(\mathrm{BE}, \mathrm{C}, \mathrm{DB}, \mathrm{avail}) = 99$.

4.1.3 Capacity and budget. Let Nodes be the set of nodes of the infrastructure. We represent the *node capacity* with a function $nodeCap : \mathsf{Nodes} \times \mathsf{Res} \to \mathbb{R}$ such that $nodeCap(n, r)$ is the maximum amount of resource r available at node n. Similarly, $linkCap : \mathsf{Nodes} \times \mathsf{Nodes} \times \mathsf{Res} \to \mathbb{R}$ models the *link capacity*, i.e., $linkCap(r, n, n')$ is the maximum amount of r that the link between n and n' can handle.

For instance, in Theorem 1.1 we have $\mathsf{Nodes} = \{n_1, n_2, n_3\}$ with $nodeCap(n_1, \mathrm{CPU}) = nodeCap(n_2, \mathrm{CPU}) = 4$, while $nodeCap(n_3, \mathrm{CPU}) = 16$. The link capacities are: $linkCap(n_1, n_2, \mathrm{latency}) = 10$, $linkCap(n_2, n_3, \mathrm{latency}) = 20$, $linkCap(n_1, n_2, \mathrm{avail}) = 98$, and $linkCap(n_2, n_3, \mathrm{avail}) = 99$.

Since we consider both monetary and energy budgets, we formalise the *budget* requirements with two functions. The first, $cost : \mathsf{Nodes} \times \mathsf{Res} \to \mathbb{R}$, returns the unit cost, in some currency, of using a resource deployed on a given node. The second, $carb : \mathsf{Nodes} \times \mathsf{Res} \to \mathbb{R}$, estimates the carbon emission per unit of a resource on a given node.

For ease of reading, Table 1 summarizes the parameters introduced in this section.

4.2 Variables

We now describe the "core" of our model, where we adopt *binary* decision variables to determine if a component is deployed in a specific flavour on a specific node. Precisely, we define $|\mathsf{Comps}|$ matrices $\mathcal{D}^c$ of $|Flav(c)| \times |\mathsf{Nodes}|$

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
NRes	Subset of $\mathbb{N}$	Set of non-consumable resources
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 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

Table 1. Input parameters of FREEDA model.

binary variables where, for each $c \in \text{Comps}$, the rows of $\mathcal{D}^c$ represent the flavours of c, and the columns of $\mathcal{D}^c$ the infrastructure nodes, such that for each $i \in \mathit{Flav}(c)$ and $j \in \text{Nodes}$:

$$\mathcal{D}^c_{i,j} = \begin{cases} 1 & \text{if component } c \text{ is deployed in flavour } i \text{ on node } j \\ 0 & \text{otherwise.} \end{cases}$$

For instance, in Theorem 1.1, we have $\mathcal{D}^{\text{FE}}, \mathcal{D}^{\text{BE}} \in \{0,1\}^{2\times 3}$ while $\mathcal{D}^{\text{DB}} \in \{0,1\}^{1\times 3}$.

This approach facilitates the adoption of different solving technologies, e.g., we can use these variables as is in a mixed-integer linear programming (MIP) model or consider them as Boolean variables for a SAT problem.

4.3 Constraints

Constraints are relations over the variables, defining the feasible solutions and encoding the deployment requirements. First of all, we must enforce that each component $c \in \text{Comps}$ is deployed in at most one flavour, on at most one node:

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

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. Thanks to this constraint, for each $c \in \mathsf{Comps}$ we can introduce an auxiliary variable:

$$node_c = \sum_{i \in Flav(c), j \in \mathsf{Nodes}} j \cdot \mathcal{D}^c_{i,j}$$

representing the node where c is possibly deployed, i.e., $node_c > 0$ if and only if c is deployed on $node_c$. Hence, we can impose that each component $m \in \mathsf{MustComps}$ must be deployed by enforcing:

$$\forall m \in \mathsf{MustComps} : \; node_m > 0 \tag{2}$$

Note that $node_m > 0$ if and only if $\sum \mathcal{D}^m_{i,j} = 1$. However, adding the *redundant* constraint $\sum \mathcal{D}^m_{i,j} = 1$ could still be beneficial, as the constraint solver may not be able to infer this information.

An important requirement is that, if component c is deployed in some flavour i, for each pair $(c', i') \in Uses(c, i)$ component c' must be deployed in a flavour $k \in Flav(c)$ at least as powerful as i':

$$\forall c \in \mathsf{Comps}, \forall i \in Flav(c), \forall (c', i') \in Uses(c, i) :$$
$$\sum_{j \in \mathsf{Nodes}} \mathcal{D}^c_{i,j} \; \leq \sum_{\substack{k \in Flav(c') \; s.t. \; k \succeq_{c'} i', \\ j \in \mathsf{Nodes}}} \mathcal{D}^{c'}_{k,j} \tag{3}$$

Moreover, we forbid the deployment of "isolated" nodes: if a component c not in MustComps is deployed, then there must be at least one component c' that requires c deployed in some flavour i. Formally:

$$\forall c \in \mathsf{Comps} \setminus \mathsf{MustComps} : \sum_{\substack{i \in Flav(c) \\ j \in \mathsf{Nodes}}} \mathcal{D}^c_{i,j} \; \leq \sum_{\substack{(c', i') \in mayUse(c) \\ j \in \mathsf{Nodes}}} \mathcal{D}^{c'}_{i',j} \tag{4}$$

In Theorem 1.1, from Eq. (1), we have that $\sum_{i,j} \mathcal{D}^c_{i,j} \leq 1$ for each $c \in \{\mathrm{FE}, \mathrm{BE}, \mathrm{DB}\}$, $i \in Flav(c)$, and $j \in \mathsf{Nodes}$. Assuming $\mathsf{MustComps} = \{\mathrm{FE}\}$, from Eq. (2) we have $\sum_{i,j} \mathcal{D}^{\mathrm{FE}}_{i,j} = 1$ and $node_{\mathrm{FE}} > 0$, so Eq. (3) becomes $1 \leq \sum_{i,j} \mathcal{D}^{\mathrm{BE}}_{i,j}$ and, therefore, by Eq. (1), $\sum_{i,j} \mathcal{D}^{\mathrm{BE}}_{i,j} = 1$ which means $node_{\mathrm{BE}} > 0$ as expected: the backend is always deployed regardless of the frontend flavour. Note that these constraints make Eq. (4) redundant for the backend because it enforces the frontend to be deployed in some flavour, which is subsumed by $node_{\mathrm{FE}} > 0$.

However, Eq. (4) is crucial to enforce the deployment of the database only when the backend is deployed in flavour cloud, thus avoid deploying the database when not needed, i.e., if the backend is deployed in flavour edge. Formally, this translates into: $\mathcal{D}^{\mathrm{DB}}_{\mathrm{S},n_1} + \mathcal{D}^{\mathrm{DB}}_{\mathrm{S},n_2} + \mathcal{D}^{\mathrm{DB}}_{\mathrm{S},n_3} \leq \mathcal{D}^{\mathrm{BE}}_{\mathrm{C},n_1} + \mathcal{D}^{\mathrm{BE}}_{\mathrm{C},n_2} + \mathcal{D}^{\mathrm{BE}}_{\mathrm{C},n_3}$. This means that if $\mathcal{D}^{\mathrm{DB}}_{\mathrm{S},n_1} + \mathcal{D}^{\mathrm{DB}}_{\mathrm{S},n_2} + \mathcal{D}^{\mathrm{DB}}_{\mathrm{S},n_3} = 1$ (i.e., DB deployed in flavour S on some node) then it must be $\mathcal{D}^{\mathrm{BE}}_{\mathrm{C},n_1} + \mathcal{D}^{\mathrm{BE}}_{\mathrm{C},n_2} + \mathcal{D}^{\mathrm{BE}}_{\mathrm{C},n_3} = 1$ (BE deployed in flavour C). On the other hand, if $\mathcal{D}^{\mathrm{DB}}_{\mathrm{S},n_1} + \mathcal{D}^{\mathrm{DB}}_{\mathrm{S},n_2} + \mathcal{D}^{\mathrm{DB}}_{\mathrm{S},n_3} = 0$ (i.e., DB not deployed) then Eq. (4) is subsumed—which happens when both the frontend and the backend are in flavour edge.

4.3.1 Component requirements. If a component c deployed in flavour i requires a certain amount of resource r, then $node_c$ must have capacity for r:

$$\forall c \in \mathsf{Comps}, \forall i \in Flav(c), \forall r \in resReq(c, i) :$$
$$node_c > 0 \implies comReq(c, i, r) \cdot \mathcal{D}^c_{i, node_c} \leq nodeCap(node_c, r) \tag{5}$$

Note that, unlike Eq. (1)–Eq. (4), this is not a linear formulation because $node_c$ is a variable whose value is generally unknown *a priori*. However, paradigms like Constraint Programming (CP) or Satisfiability Modulo Theory (SMT) can easily handle this formulation. Moreover, Eq. (5) can be easily *linearised*—i.e., transformed into an equisatisfiable formulation only including linear constraints—at the expense of adding more inequalities.

For *consumable* resources only, we must guarantee that a node fulfills the resource requirements for *all* the components deployed on it. Let CR_j be the set of all the consumable resources available on node j. We impose that:

$$\forall j \in \mathrm{Nodes}, \forall r \in \mathrm{CR}_j : \sum_{\substack{c \in \mathrm{Comps}, \\ i \in Flav(c) : r \in resReq(c,i)}} comReq(c,i,r) \cdot \mathcal{D}^c_{i,j} \leq nodeCap(j,r) \tag{6}$$

In this way, we ensure that each node j has a sufficient quantity of a certain resource r to meet the demands of all the components deployed on it.

In Theorem 1.1, Eq. (5) for $c = \mathrm{DB}$ and $r = \mathrm{CPU}$ becomes $node_{\mathrm{DB}} > 0 \Rightarrow \mathcal{D}^{\mathrm{DB}}_{\mathrm{S},node_{\mathrm{DB}}} \leq nodeCap(node_{\mathrm{DB}}, \mathrm{CPU})$. Because the CPU is consumable, we must also ensure that a node has enough CPUs for all the components it hosts, i.e., $\mathcal{D}^{FE}_{\mathrm{E},j} + 2\mathcal{D}^{FE}_{\mathrm{C},j} + \mathcal{D}^{BE}_{\mathrm{E},j} + 2\mathcal{D}^{BE}_{\mathrm{C},j} + \mathcal{D}^{DB}_{\mathrm{S},j} \leq nodeCap(j, \mathrm{CPU})$ for $j \in \{n_1, n_2, n_3\}$.

4.3.2 Dependency requirements. To ensure the satisfaction of the dependency requirements between interdependent components, we impose the following:[1]

$$\forall c' \in \mathrm{Comps}, \forall (c,i) \in mayUse(c'), \forall r \in resReq(c,i,c') : depReq(c,i,c',r) \cdot \sum_{j \in \mathrm{Nodes}} \mathcal{D}^c_{i,j} \leq linkCap(node_c, node_{c'}, r) \tag{7}$$

Note that this constraint is only relevant when the "source" component c is deployed in flavour i on some node, which involves that also the "target" component c' must be deployed, being $(c', i') \in Uses(c,i)$. Instead, if c is not deployed, then $\sum_{j \in \mathrm{Nodes}} \mathcal{D}^c_{i,j} = 0$, so Eq. (7) becomes subsumed being $linkCap(node_c, node_{c'}, r) \geq 0$.

Similarly to Eq. (5), also Eq. (7) is non-linear, but it can be linearised by introducing additional variables and constraints, thereby increasing the size of the problem.

For instance, Theorem 1.1 involves the constraint $99 \cdot \sum_{j \in \mathrm{Nodes}} \mathcal{D}^{\mathrm{BE}}_{\mathrm{C},j} \leq linkCap(node_{\mathrm{BE}}, node_{\mathrm{DB}}, \mathrm{avail})$ because $depReq(\mathrm{BE}, \mathrm{C}, \mathrm{DB}, \mathrm{avail}) = 99$, which implies that the frontend and the database cannot be deployed on nodes n_1 and n_2, being $linkCap(n_1, n_2, \mathrm{avail}) = 98$.

4.3.3 Budget requirements. The following variables denote, respectively, the total cost and carbon emission of the deployment:

$$tot_{cost} = \sum_{\substack{c \in \mathrm{Comps}, i \in Flav(c), \\ r \in resReq(c,i), j \in \mathrm{Nodes}}} comReq(c,i,r) \cdot cost(j,r) \cdot \mathcal{D}^c_{i,j} \tag{8}$$

$$tot_{carb} = \sum_{\substack{c \in \mathrm{Comps}, i \in Flav(c), \\ r \in resReq(c,i), j \in \mathrm{Nodes}}} comReq(c,i,r) \cdot carb(j,r) \cdot \mathcal{D}^c_{i,j} \tag{9}$$

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

4.4 Objective function

To fulfil the definition of optimal deployment, we need to define what "optimal" means for us. Multiple objective functions can be formulated. For example, a "*cost-oriented*" deployment, aiming to minimise tot_{cost}, an "*environmental-friendly*" deployment, minimising tot_{carb}, or a *hybrid* approach that minimises $\gamma \cdot tot_{cost} + \varepsilon \cdot tot_{carb}$ with γ and ε user-defined parameters. While the latter is more flexible, it is crucial to carefully tune γ and ε to

[1]For better readability, we assume non-consumable resources. For consumable resources, the same reasoning of Eq. (6) can be directly applied.

reflect the relative importance of cost and carbon factors, typically involving different units of measurement. Furthermore, minimising tot_{cost} and/or tot_{carb} will likely result in a *minimal* deployment with the least number of components in their "less powerful" flavour (e.g., in Theorem 1.1, frontend and backend in flavour edge, and no database deployed).

These approaches, however, would lead to a lower QoS for the end user, who would not benefit from the performance advantages of more powerful flavours. Instead, FREEDA employs budgets β_{cost} and β_{carb} as constraints on tot_{cost} and tot_{carb} respectively. By using this approach, FREEDA allows for a more balanced deployment strategy that meets the requirements without prioritizing cost or carbon reduction at the expense of the end user. This ensures that the deployment is not only resource-aware but also capable of reaching high levels of performances.

To avoid these issues, in this work, we focus on the *importance* of the flavours. We define the objective function to *maximise* the importance of deployed component flavours as:

$$\sum_{\substack{c \in \text{Comps}, \\ i \in Flav(c)}} imp(c, i) \cdot \left(\sum_{j \in \text{Nodes}} \mathcal{D}^c_{i,j} \right) \tag{10}$$

Function *imp* defines the goal of our deployment: deploy the flavours with the highest importance for the users while respecting cost and carbon emissions constraints. To synthesise *imp*, the most intuitive approach is to assign an incremental importance value to each flavour of a given component c, based on $\preceq_c$. For example, the least powerful flavour could be assigned a value of 1, the second-least powerful a value of 2, and so on. However, this approach might lead to undesirable deployment solutions.

For example, suppose that a component c_1 has only one flavour f with high importance (e.g., $imp(c_1, f) = 3$) and does not use any other component: $Uses(c_1, f) = \emptyset$. Suppose that also components c_2 and c_3 have only one flavour g with medium importance ($imp(c_2, g) = imp(c_3, g) = 2$) and that $Uses(c_2, g) = \{(c_3, h)\}$, i.e., when c_2 is deployed in flavour g, then also c_3 must be deployed in a flavour at least as powerful as h. At this point, if at most two components can be deployed, e.g., due to budget constraints, the deployment of c_2 and c_3 will be chosen instead of deploying c_1 in its more powerful flavour because $2 + 2 > 3$.

A possible workaround is to leave '*gaps*' between the importance values, e.g., high importance = 7, medium = 3, low = 1. The question then becomes: how to define these gaps? A plausible option is to define $\lambda > 0$ *priority levels* as follows. For $i = 1, \ldots, \lambda$ let F_i be the set of all flavours having priority i (1 is the lowest priority, and higher values means higher priority) and let $n_i = |F_i|$. Then, for each $c \in \text{Comps}$ and $f \in \text{Flavours}(c)$, we define:

$$imp(c, f) = \begin{cases} 1 & \text{if } f \in F_1 \\ \prod_{j=1}^{i-1}(n_j + 1) & \text{if } f \in F_i \text{ with } i > 1 \end{cases} \tag{11}$$

For example, suppose we have $\lambda = 3$ priority levels and $n_1 = 5$ flavours having priority 1 (low), $n_2 = 3$ flavours with priority 2 (medium) and the rest with priority 3 (high). We have:

- $imp(c, f) = 1$, if $f \in F_1$
- $imp(c, f) = n_1 + 1 = 5 + 1 = 6$, if $f \in F_2$
- $imp(c, f) = (n_1 + 1)(n_2 + 1) = 6 \cdot 4 = 24$, if $f \in F_3$

In this way, each flavour with priority 3 weighs one plus the sum of all flavours with priority 1 and 2, and each flavour with priority 2 weighs one plus the sum of all those with priority 1. This formulation forces the deployment of the high-priority flavours, followed by the mid-priority and then the low-priority ones. Note that this approach works well with a reasonable number of priority levels, as $imp(c, f)$ grows exponentially according to λ.

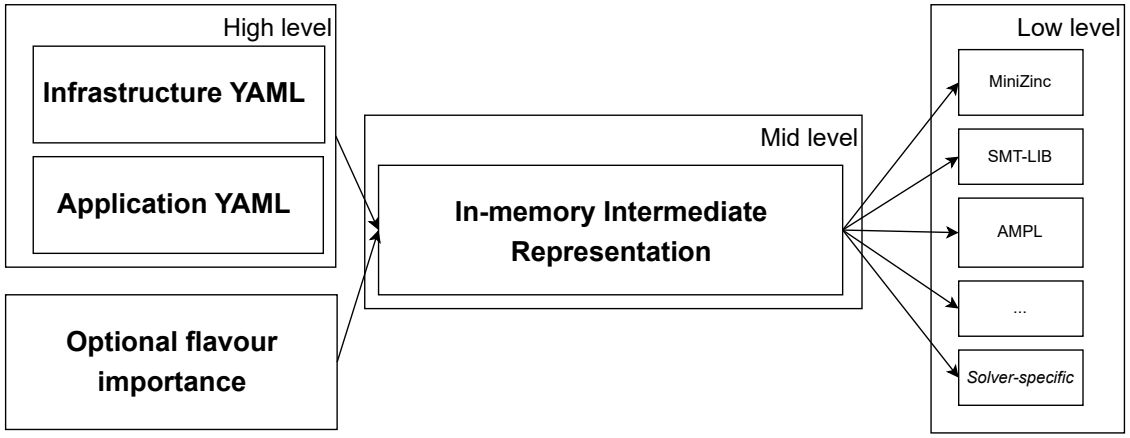


Fig. 2. Hierarchical structure of FREEDA architecture.

5 Implementation

In this section, we describe the implementation of the FREEDA framework. Our approach uses a hierarchical structure, depicted in Fig. 2, to streamline the representation and processing of deployment architectures. The framework provides distinct representations to facilitate a structured and flexible transition from human-friendly descriptions to solver-executable models. This design effectively balances expressiveness, pre-processing robustness, and compatibility with diverse solving technologies. Fig. 2 depicts the hierarchical structure we propose. The implementation is an open-source, publicly available project [35].

The *high-level* module allows the user to specify the deployment architecture—components, requirements, and infrastructure—in the YAML language, as described in Section 3.

The YAML specification is then parsed into an in-memory *mid-level* representation. The parse process, implemented in Python 3.11, encompasses syntactical and semantic checks to ensure that the input is well-formed and adheres to the specifications outlined in Section 3. For instance, the parser ensures that, when defining a quality of service for dependencies between nodes, both source node and target node must be specified in the node-capabilities YAML section (see Section 3.3). Additionally, when specifying component requirements, resources in *flavour-specific* and *common* sections must be a disjoint set. The purpose of this parsing phase is to generate an optimised representation of a specific problem instance, i.e., the parameters described in Section 4.1.1 are extracted and instantiated from the YAML specifications. Notably, the importance values of each flavour are assigned during this phase. By default, incremental importance values are system-assigned to the flavours of a given component: the least powerful flavour is assigned a value of 1, the second-least 2, and so forth. However, our implementation allows the user to override the importance values, provided they respect the invariant $f \preceq_c f' \Leftrightarrow imp(c, f) \leq imp(c, f')$. Users can also select from predefined options, such as reverse incremental importance (where a specified value k is assigned to the most powerful flavour, $k-1$ to the second-most powerful, and so on) or the priority-based approach formalised in Eq. (11).

The intermediate representation is not yet in a solver-executable format; however, one can perform additional checks and optimisations at this stage. Specifically, one could introduce instance-specific pre-solving and *symmetry-breaking*. For example, if the infrastructure forms a complete graph with all nodes and links having identical capabilities, it becomes feasible to arbitrarily sort the nodes assigned to each component. Nevertheless, this represents a corner case. In general, automatically determining an effective and *sound* symmetry-breaking

strategy is a challenging task, given the arbitrary number of resources and constraints involved in the problem. Therefore, we leave the exploration and development of symmetry-breaking strategies as future work.

From the same mid-level representation, different *low-level* executable models can be derived, depending on the underlying solver(s) available. Indeed, the same model can be tackled by different solvers, supporting diverse underlying technologies and languages. For example, SMT solvers require a SMT-LIB [10] encoding, while most MIP solvers support AMPL [32] language. Using a solver-independent language like SMT-LIB or AMPL provides greater flexibility as the same low-level representation can be solved by any solver supporting that language. However, for larger problem instances, reducing the parsing time may necessitate generating a *solver-specific* encoding directly. Currently, FREEDA uses the *MiniZinc* language [52] to encode the low-level representation. MiniZinc is the most common language for modeling CP problems. It offers a high level of expressiveness, enabling the straightforward modeling of complex constraints. It is solver-independent (its motto is "*Model once, solve anywhere*") and supports a wide range of backend solvers—not only CP solvers, e.g., MIP solvers like Highs [43] or hybrid solvers like OR-Tools [38]. This choice allows to exploit diverse solving technologies without being tied to a specific solver.

Another feature of MiniZinc is the separation between model and data: the same parametric model can be instantiated with different input data to represent specific instances of the problem. MiniZinc's bundle includes a built-in compiler that compiles model and data into a restricted language, called *FlatZinc*. This step, transparent to the user, enables solver-specific redefinitions (e.g., constraints Eq. (5) and Eq. (7) can be kept as is, or linearised to better accommodate a MIP solver) and facilitates the parsing into the solver-specific language. Thanks to this feature, FREEDA defines only a single MiniZinc model (`.mzn` format) implementing the variables, the constraints, and the objective function of Section 4. The corresponding data is generated by transforming the mid-level representation into a MiniZinc data file (`.dzn` format) representing the parameters explained in Section 4.1.1. Model and data are then jointly compiled into FlatZinc and passed to the underlying solver.

Listing 4 shows the different MiniZinc output we get for Theorem 1.1 when varying the cost budget, while maintaining the carbon budget β_{carb} fixed at 500. For example, if $\beta_{cost} = 600$ as in Listing 2, we can only get a minimal deployment, as selecting more powerful flavours would exceed the budget. If we increase the budget to 850, we can afford a backend in the cloud flavour and a database, with total cost 812, but not a frontend in the edge flavour. The "maximum deployment", where all components are in their most powerful flavours, requires $\beta_{cost} \geq 932$.

6 Evaluation

In this section, we present a comprehensive evaluation of FREEDA performance. We test a plethora of deployment cases through a two-part analysis. First, in Section 6.1, we analyse FREEDA's scalability across various infrastructure topologies and solver configurations. Then, in Section 6.2, we compare FREEDA and Zephyrus, using a custom test generation framework designed to create equivalent configurations while bridging their architectural differences, such as Zephyrus' inability to handle non-complete topologies and flavours.

We run each test on Intel Core i5-4590 3.30GHz machines with 8 GB of RAM. All the data and source code we used in these experiments is publicly available at [34].

6.1 Scalability

The objective of our scalability analysis is to profile the performances of FREEDA by considering different deployment problems of increasing size. Specifically, our first research question is: how does FREEDA behave when increasing the number of components and/or nodes?

```
            %%%%% Cost budget = 600 %%%%%
Component frontend deployed in flavour edge on node n3.
Component backend deployed in flavour edge on node n3.
Component database not deployed.
Objective value: 2
Total cost: 256
Total carb: 50
            %%%%% Cost budget = 850 %%%%%
Component frontend deployed in flavour edge on node n3.
Component backend deployed in flavour cloud on node n3.
Component database deployed in flavour standard on node n3.
Objective value: 4
Total cost: 812
Total carb: 100
            %%%%% Cost budget = 1000 %%%%%
Component frontend deployed in flavour cloud on node n3.
Component backend deployed in flavour cloud on node n3.
Component database deployed in flavour standard on node n3.
Objective value: 5
Total cost: 932
Total carb: 125
```

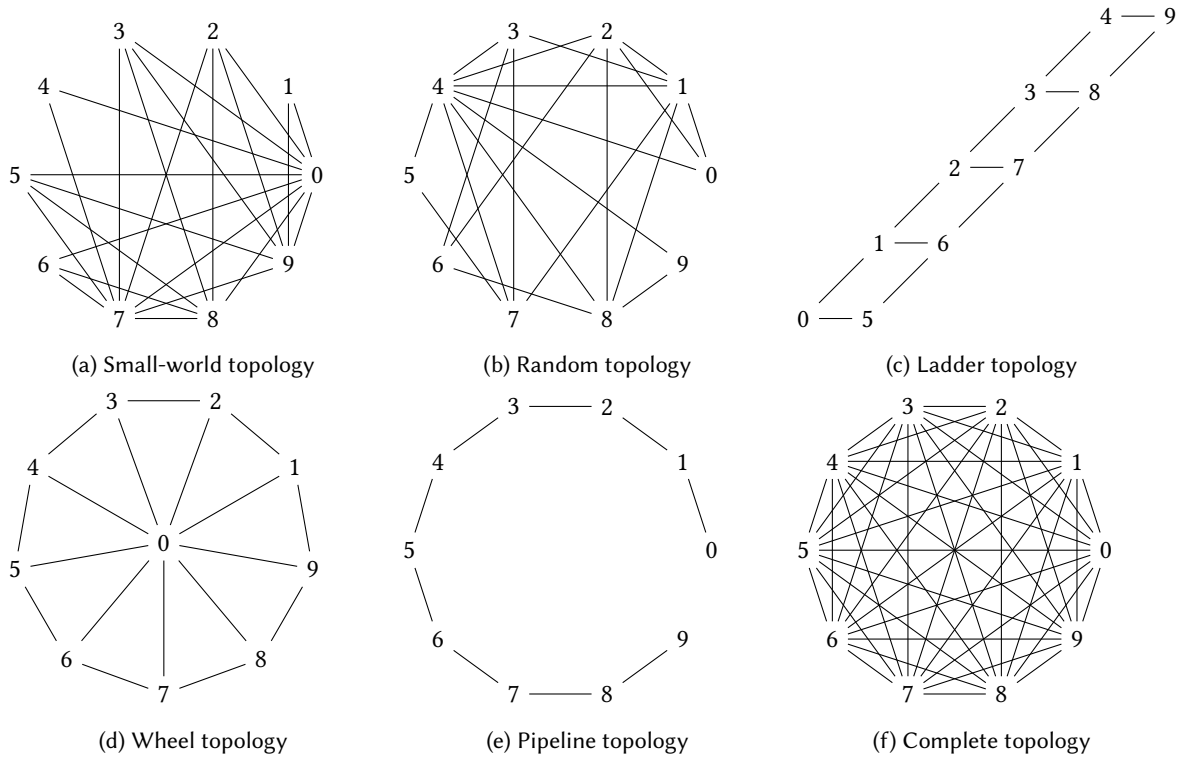
Listing 4. FREEDA's deployments of Theorem 1.1 with different cost budgets.

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. Also, we consider 3 different components' topologies commonly observed in microservice[2] architectures [54]: pipeline (each node is connected to its immediate predecessor and successor, forming a linear sequence), small-world [9] (typically including cliques and sub-networks connecting almost every pair of nodes), and random [25] (edges randomly generated with uniform probability). We also evaluate 5 infrastructure topologies: complete, small-world, random, ladder (two pipelines of nodes, with one additional edge per node connecting nodes in different pipelines) and wheel (a ring with a hub node connected to all the others). Fig. 3 depicts the shape of the above topologies. Therefore, in total, we evaluate $(40 - 3 + 1)^2 \cdot 3 \cdot 5 = 21660$ different deployment scenarios.

For each component in every scenario, we randomly generate up to three different flavours, assigning default incremental importance (as explained in Section 5). We fix the number of resources to five and randomly determine, with uniform probability, whether a resource is consumable or not. To avoid trivially unfeasible deployments, we impose that each individual component can be deployed on at least one node. The carbon and cost budget are set to enable each individual component to be deployed (if needed) in its largest flavour. We randomly determine whether a component belongs to MustComps.

We repeat each (random) scenario generation 3 times to mitigate the side effects of randomness, hence we solve $21660 \cdot 3 = 64980$ instances with 4 different solvers: Gecode [14] and Chuffed [15] (CP solvers), Highs [43] (MIP solver), and OR-Tools [38] (employing a hybrid CP-SAT-MIP approach). Thus, we conduct a total of

[2]Microservices are one of the main state-of-the-art architectural patterns in Cloud systems [23].



(a) Small-world topology (b) Random topology (c) Ladder topology

(d) Wheel topology (e) Pipeline topology (f) Complete topology

Fig. 3. Examples of topologies.

solver	solved	error	avg all	avg solved	min	max	first	last	flat	score
OR-Tools	100.0%	0.0%	1.34s	1.34s	0.41s	56.56s	1.22s	1.31s	1.03s	154489.28
Highs	99.07%	0.93%	N/A	1.39s	0.43s	300.0s	1.34s	1.36s	1.06s	150358.60
Gecode	27.71%	0.0%	221.58s	20.23s	0.42s	300.0s	1.18s	31.61s	1.08s	48876.90
Chuffed	20.0%	0.0%	242.73s	18.74s	0.43s	300.0s	1.22s	5.76s	1.16s	36155.23

Table 2. FREEDA performance with different solvers.

$64980 \cdot 4 = 259920$ experiments. Because these problems are NP-hard, we set a timeout of 300 seconds. We say that a solver solves a problem if it solves it to optimality or proves its unsatisfiability. If a solver cannot solve a problem, its runtime is set to the timeout value, without any additional penalties.

We report in Table 2 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 is the only solver that provides unsound answers. Notably, OR-Tools solves all the instances.

Unsatisfiable problems are around the 0.02% of all scenarios. Note that we cannot know *a priori* whether a problem we generate is 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 does 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 solves 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.

Columns 'first' and 'last' show the average time to find the first and the last solution respectively. The 'first' column gives an idea of how much the solver is responsive. The 'last' column indicates how long a solver takes to prove that the incumbent solution is optimal.

Notably, Chuffed flattening time, being a bit slower than Gecode, on average penalises its finding of the first solution. Furthermore, the nature of solvers matters: CP solvers like Chuffed and Gecode excel at quickly finding an initial solution, but sometimes struggle to improve it, or to prove its optimality. However, performance improvements may be achievable through an ablation study of different search heuristics, the adoption of large neighborhood search, or the addition of symmetry-breaking constraints.

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 FREEDA model, having a linear objective function and predominantly linear constraints (as detailed in Section 4), hence the higher solving percentage.

The 'score' column confirms the above observations, showing the cumulative MiniZinc Challenge score [58]. In the score, each pair of solvers s and s' is evaluated on every instance. If s gives a better answer than s', it scores 1 point; if it gives a worse, incorrect or 'unknown' answer it scores 0 points; otherwise the score is based on the runtime: if t and t' are the corresponding solving times, quantised to seconds, then s scores $\frac{t'}{t+t'}$ and s' scores $\frac{t}{t+t'}$ (if $t = t' = 0$ they both score 0.5).[3]

To get more insights on the impact of the topology on the solvers' performance, in Fig. 4, we plot the solving times of each solver on every test instance—sorted by increasing runtime—for all pairs of component-infrastructure topologies. We observe that, as previously noted, Chuffed and Gecode solve fewer problems than Highs and OR-Tools. Overall, the results suggest that the performance of the solvers is not strongly affected by variations in infrastructure or application topologies, as all plots in Fig. 4 exhibit a similar behaviour.

The empirical results shown so far demonstrate that, in general, FREEDA performs well when increasing the number of components and nodes, particularly when leveraging a solver based on MIP technology. However, we conduct a deeper evaluation to better understand the relationship between the number of components and nodes and their impact on solving time.

Since the deployment problem is NP-complete, even moderate increases in problem size or constraint density can lead to significant computational overhead. Therefore, evaluating how solving time changes with different configurations provides insights into the performance boundaries of the system.

We hypothesize that both the number of components and nodes jointly influence the solving time, with the ratio $r = \dfrac{|\text{Comps}|}{|\text{Nodes}|}$ being the most critical factor. When r is too small, the problem is likely *under-constrained*: with relatively few components and many nodes, placing the components becomes an easier task. Conversely,

[3]The interested reader can find the comprehensive explanation of the scoring in the Rules of the MiniZinc Challenge [51].

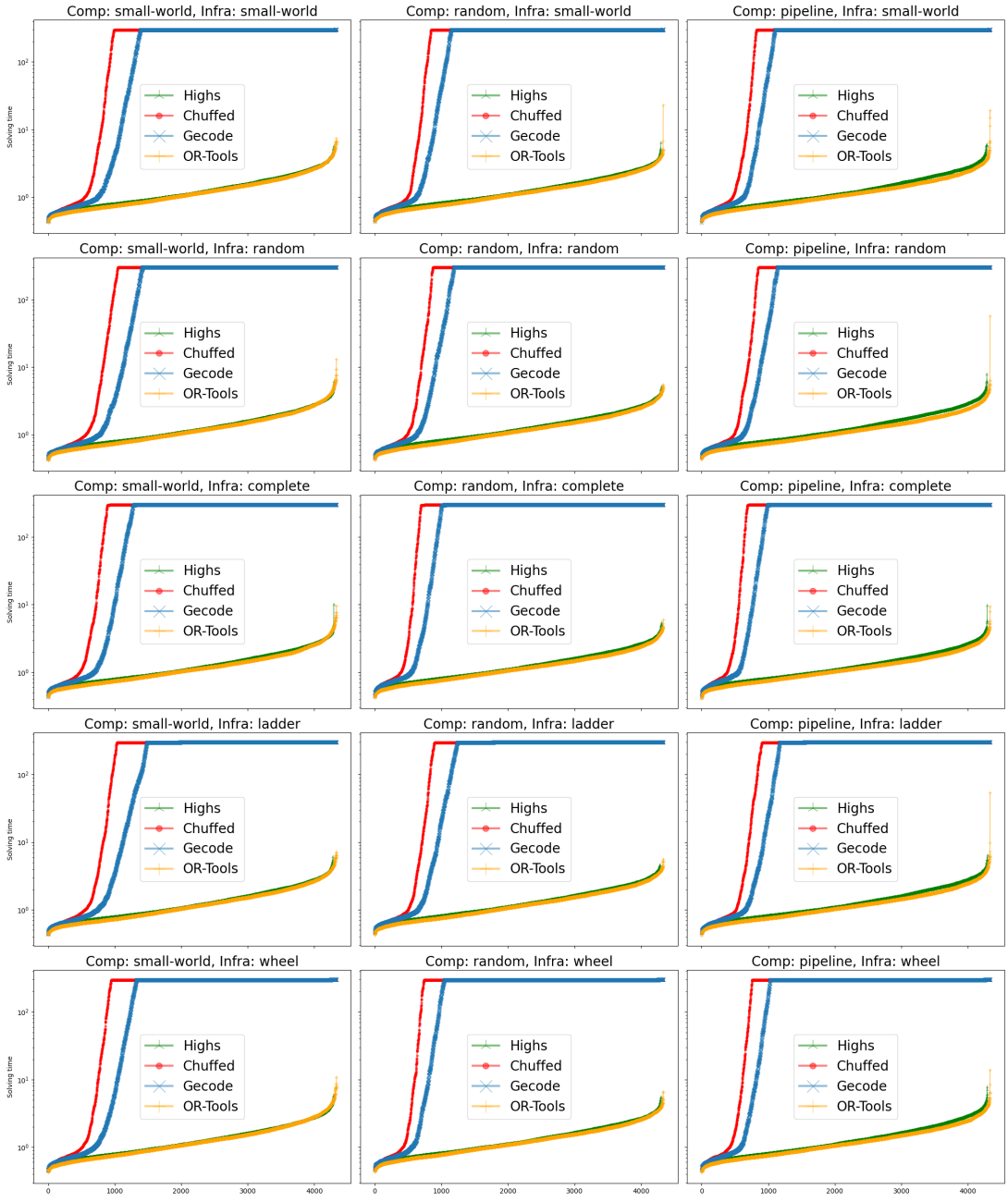


Fig. 4. Solving time in seconds (note the logarithmic scale) for each topology combination.

when r is too large, the problem becomes *over-constrained*: with many components and few nodes, the problem may be trivially infeasible.

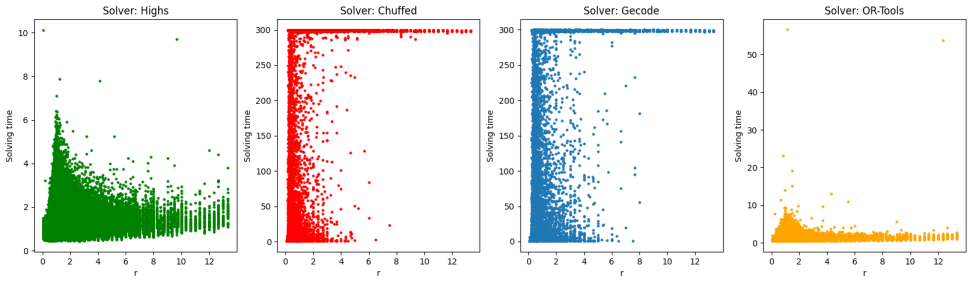


Fig. 5. Scatter plot per solver of solving times over the ratio r = |Comps| / |Nodes|.

To empirically verify this effect, in Fig. 5, we plot the solving times for all the solvers while varying r. The results show a clear pattern: when r is close to one, i.e., when the number of components is approximately equal to the number of nodes, the problems get more challenging. This observation aligns with our hypothesis. However, the reverse implication does not always hold, particularly for solvers like Chuffed and Gecode: solving a problem to optimality can still be difficult even when r is not close to one. The reason is probably that, when $r \approx 1$ and the problem is satisfiable, the solver might navigate a narrow set of feasible choices for assigning components to nodes, often encountering repeated failures that slow down the resolution.

Summarising, the experimental results presented above demonstrate that FREEDA performs reasonably well as the number of components and nodes increases, despite the NP-hard nature of these problems. As expected, performance depends significantly on the solving technology employed. While the current solving times may appear high for real-time applications, they remain practical when considering that deployment and infrastructure changes typically dominate the overall system reconfiguration time—for instance, container deployment can take 2–3 minutes on average, while VM provisioning often requires 5–10 minutes even in modern cloud environments [36, 55]. It is important to note, however, that these results should be regarded as a baseline, as we used the solvers as black boxes. A proper fine-tuning of their parameters would likely lead to improved performance. Moreover, employing a *portfolio* of two or more solvers, using diverse technologies, could help leverage the complementary strengths of each solver, potentially enhancing overall performance and solution quality.

6.2 Comparison with Zephyrus

After assessing the scalability of FREEDA, we compare its performance with related state of the art, specifically, *Zephyrus* [17].

Zephyrus is designed to address deployment optimisation problems for cloud applications. The deployment is specified through three inputs: (a) the available software components and their requirements, modelled using the *Aeolus* component model [18], where components have provide- and require- ports to represent interface dependencies; (b) the available virtual machines (named locations) and their resources, detailing attributes like memory capacity and associated costs; and (c) deployment constraints, which define specific requirements for the configuration, such as enforcing the deployment of at least one target component. The primary purpose of Zephyrus is to automatically generate optimal deployment configurations that meet all component requirements and user-defined constraints while minimizing costs. Like FREEDA, the tool transforms the deployment problem into a constraint optimisation problem encoded in MiniZinc. Zephyrus is one of the pioneering tools to formalize automated deployment optimisation at a level similar to ours and has been successfully applied in multiple industrial case studies [7, 16, 21]. In this work, we focus on the last version of the tool, Zephyrus2 [2, 11].

6.2.1 Establishing a common ground. There are several differences between Zephyrus and FREEDA in how they handle component deployment and resource management, which we summarise in Table 3. Since we want to compare them, in this section, we establish a common ground for the comparison.

Feature	**Zephyrus**	**FREEDA**
Component Flavours	×	✓
Dependencies (modelled with)	Require/offer ports	Topological dependency graph
Component Replication	✓	×
Non-consumable Resources	×	✓
Link Resources	×	✓
Cost Model	Per location	Per unit of resource
Network Topology	Complete (graph)	Any kind
Minimum Arch. Components	Requires at least 4	No minimum specified
Carbon Emissions	×	✓

Table 3. Comparison of features between Zephyrus and FREEDA (×: not supported, ✓: supported)

To run our experiments, we first generate the YAML specification in memory (i.e., without producing a `.yaml` file) for each test instance, as described in Section 3. Then, by leveraging the hierarchical structure of Fig. 2, we compile each YAML into two MiniZinc data files (since they both use MiniZinc, FREEDA and Zephyrus share the same data format, cf. Section 5).

A first significant difference between Zephyrus and FREEDA is that Zephyrus ignores the concept of flavour. Thus, to make a valid comparison, we restrict each FREEDA component to have only one flavour.

Zephyrus handles *types* of components, i.e., components with attributes like replication policies. FREEDA instead model components without replication. Thus, for each Zephyrus' *type* we limit to one the number of replications, so they are comparable to a FREEDA component-flavour pair. We impose this property by adding a constraint to the MiniZinc model of Zephyrus.

Zephyrus establishes dependencies between components by specifying the ports that each component requires. To replicate the behaviour of FREEDA's *Uses* function in Zephyrus, we generate each component with: (a) one require port for each component it *Uses* and (b) one unique port for each component that requires it.

Contrary to FREEDA, Zephyrus does not support non-consumable resources. Hence, we do not generate this type of resources. The same applies to resources on links: they cannot be specified in the Zephyrus model, so we do not generate these type of resources.

We generate test configurations ensuring that Zephyrus locations—hosting Zephyrus components—matches the FREEDA nodes—hosting the FREEDA components. Furthermore, Zephyrus always considers locations connected with a complete graph. Hence, we limit the generation of nodes/locations to complete topologies.

Zephyrus considers costs per locations while FREEDA considers costs per unit of resource. To make costs comparable, we generate only one (consumable) resource per deployment scenario.

Zephyrus does not take into account carbon emissions. Hence, we impose that the *carb* function (defined in Section 4.1.3) always returns zero.

To prevent the productions of "empty" deployments from Zephyrus (i.e., no component is deployed), we model FREEDA's concept of MustComps inside Zephyrus. For each component, we randomly specify whether it is a MustComps and, by adding a constraint to the MiniZinc Zephyrus model, we impose that each component flagged as MustComps has a cardinality of one.

We adapt FREEDA's objective function to minimise the total cost of the deployed components, ensuring alignment with Zephyrus' objective function.

model	solver	solved	error	avg all	avg solved	min	max	first	last	flat	score
FREEDA	OR-Tools	100.0%	0.0%	0.74s	0.74s	0.42s	3.07s	0.71s	0.73s	0.65s	24631.39
	Gecode	100.0%	0.0%	0.76s	0.76s	0.43s	3.15s	0.74s	0.75s	0.66s	24619.36
	Chuffed	100.0%	0.0%	0.76s	0.76s	0.42s	2.63s	0.69s	0.75s	0.65s	24513.30
	Highs	100.0%	0.0%	0.95s	0.95s	0.45s	8.16s	0.93s	0.93s	0.86s	22702.78
Zephyrus	Chuffed	99.8%	0.0%	19.31s	18.74s	0.13s	300.0s	14.09s	18.97s	7.01s	13173.54
	OR-Tools	95.02%	0.0%	30.74s	16.67s	0.13s	300.0s	15.51s	16.83s	6.99s	12774.97
	Highs	100.0%	0.0%	21.22s	21.22s	0.15s	147.05s	20.94s	21.1s	19.48s	10714.59
	Gecode	25.9%	21.09%	169.87s	8.05s	0.15s	300.0s	6.77s	9.97s	7.03s	3286.08

Table 4. Columns meaning are the same as the ones in Table 2. Rows are ordered by score.

Analogously to Section 6.1, we generate different configurations by varying the number of components, the number of nodes, and the components' topology. For the latter, we use the pipeline, small-world, and random topologies (see Fig. 3). Note that the infrastructure is always complete, as required by Zephyrus. Since Zephyrus requires at least four components, the smallest configuration for the test set has four components and three nodes. Then, we generate tests up to 32 components and nodes. In total, we therefore evaluated $(32 - 4 + 1) \cdot (32 - 3 + 1) \cdot 3 = 2610$ different deployment scenarios. Also in this case, we mitigate the side effects of randomness by generating 3 test configurations for each scenario, for a total of $2610 \cdot 3 = 7830$ configurations.

We run each configuration with both FREEDA's and Zephyrus's model, each using three solvers (OR-Tools [38], Chuffed [15], and Highs [43]). As in Section 6.1, we impose a timeout of 300 seconds. In total, this results in $7830 \cdot 2 \cdot 3 = 46980$ experiments.

6.2.2 Results. We present the empirical results of the comparison in Table 4 (the performance metrics are identical to those in Table 2). The data indicate that FREEDA, regardless of the solver used, successfully solves all problems ('solved' column). Zephyrus' performance is comparable, although only with Highs because it fails to solve some problems when using OR-Tools, Chuffed or Gecode. The average solving times ('avg all' and 'avg solved' columns) highlight a significant difference: the worst FREEDA performance is more than 20 times better than the best performance of Zephyrus' approach. We noted that Gecode runs out of memory on 21.09% of the scenarios. This behavior is expected, as Gecode is a *copying* solver: at each branch in the search tree, it clones the entire search state—including variables, constraints, and other metadata—into a new space. While this approach simplifies backtracking, it significantly impacts scalability when the search tree becomes large. Notably, FREEDA can solve all the instances in less than 9 seconds, regardless of the solver used (see 'max' column).

For a few easy instances, Zephyrus is faster than FREEDA. Specifically, Chuffed is slower 1787 times, Highs for 1389, OR-Tools in 1850 and Gecode in 1340 for a total of 6366 instances (10.16% of the total amount of tests). This is also reflected by Zephyrus' better 'min' time. However, Zephyrus often struggles to find an initial solution, as evidenced by the differences in the 'first' column.

One plausible explanation for the performance difference between Zephyrus and FREEDA is that Zephyrus model is "heavier", as it involves a four-dimensional matrix of variables and many logical implications. This complexity results in higher conversion times from MiniZinc to FlatZinc compared to FREEDA (as shown in the 'flat' column) and leads to larger search spaces due to the increased number of variables. Unsurprisingly, a larger search space typically results in longer solving times.

The scatter plots in log scale shown in Fig. 6 empirically confirm this observation. The upper part of Fig. 6 presents the distribution of solving times, sorted in ascending order, for all solvers and problem instances.

FREEDA's curves start higher (the reason is explained earlier—see the 'min' column in Table 4) but are soon overtaken by Zephyrus. In the lower part of Fig. 6 we show the number of variables in both models—including the auxiliary variables introduced by the flattening process. The problems are sorted by solving times. In the Gecode column, the instances for which the solver failed to compute a solution are marked in black to distinguish them from successfully solved cases. The distribution clearly shows the significant growth in the number of variables used by Zephryrus' model across all solvers.

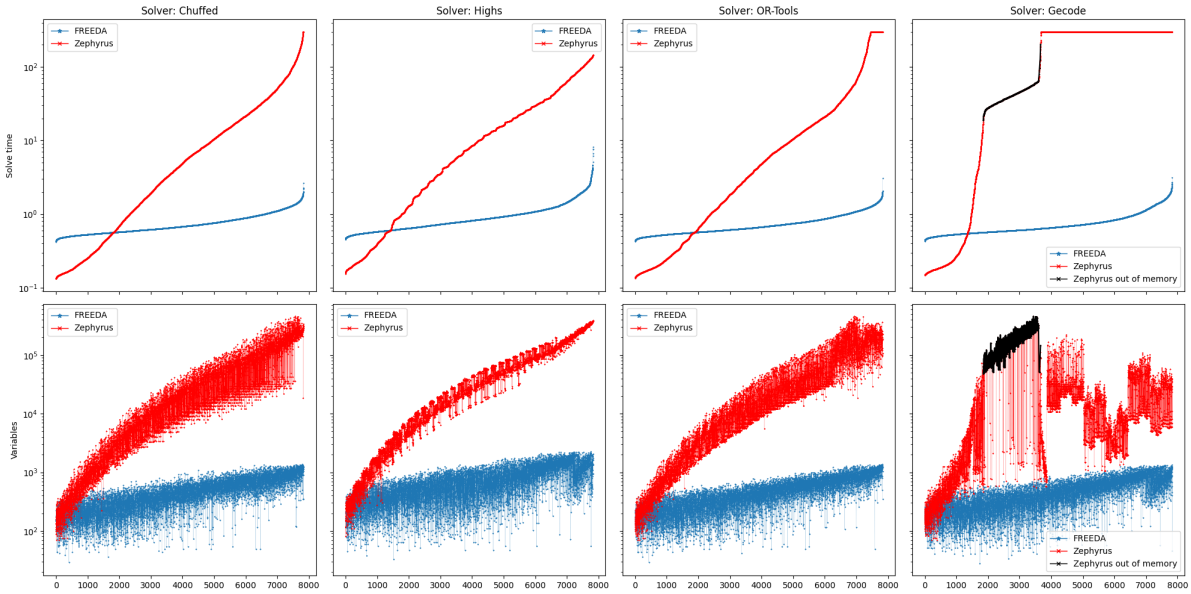


Fig. 6. Upper part: solving time in seconds for each solver on each problem. Lower part: number of variables for each problem instance.

Summarising the results of these experiments, we believe that the stark performance difference between FREEDA and Zephyrus arises from the structure of the underlying constraint model. In particular, despite the expressiveness introduced by flavours, FREEDA's model (Section 4) is more "lightweight" and allows a faster resolution for most solvers.

7 Concluding Remarks

We propose a constraint optimisation model to jointly decide which application topology to deploy and where to deploy application components, without exceeding the available cost/carbon budget, and optimising the number of components deployed in their preferred flavour. We showcase the practical feasibility of the proposed approach, by introducing an open-source MiniZinc implementation of the model and by running it over a lifelike example—by relying on state-of-the-art solvers. Furthermore, we evaluate our proposal by running comparison tests with the state-of-the-art Zephyrus cloud deployment framework, showing that, on the tested solver, Freeda is mostly better than or at least comparable to Zephyrus. We also run scalability tests to see how the solver's behaviour changes by increasing the number of deployable components and infrastructure nodes or by changing the topologies.

The problem addressed by FREEDA has a significant inherent computational complexity. Since component placement with flavour selection under constraints is an NP-complete problem, the scalability of the approach remains a challenge. While constraint optimisation provides a flexible and expressive formalism, its performance can degrade significantly as the size of the deployment grows—particularly in scenarios involving large numbers of components, flavours, and infrastructure nodes. As a future work, we plan to explore (meta-)heuristics and local search techniques (such as Large Neighborhood Search) as well as combining optimisation with Machine Learning to possible learn data patterns from different deployment scenarios.

A possible future direction concerns the use of learning mechanisms. We hypothesise that it is possible to leverage historical placement data to build a knowledge base that captures patterns of effective deployments. Such a knowledge base could be incrementally refined over time and used during execution to guide and accelerate the placement process, potentially improving scalability.

Another direction for future work is plugging the proposed optimisation model into an end-to-end toolchain that would enable the sustainable and failure-resilient deployment of Cloud-Edge applications by using an energy-optimisation module and a failure verifier, as envisioned by the FREEDA project [57]. Furthermore, we plan to conduct experiments in real-world case studies involving actual machines and hardware infrastructures. These evaluations will allow us to assess the practical applicability, performance, and robustness of FREEDA in realistic deployment scenarios.

Furthermore, we can integrate *explanations* of why a certain deployment is unfeasible or optimal through the use of specialized solvers extracting a—possibly minimal—unsatisfiable set of conflicting constraints.

Acknowledgments

This work is mainly supported by the project FREEDA (CUP: I53D23003550006), funded by the frameworks PRIN (MUR, Italy) and Next Generation EU. The work is also partly supported by PNRR - M4C2 - Investimento 1.3, Partenariato Esteso PE00000013 - "FAIR - Future Artificial Intelligence Research" - Spoke 8 "Pervasive AI"; "hOlistic Sustainable Management of distributed softWARE systems" (OSMWARE, PRA_2022_64) funded by the University of Pisa, Italy; *Energy-aware management of software applications in Cloud-IoT ecosystems* (RIC2021_PON_A18) funded by the Italian MUR over ESF REACT-EU resources through *PON Ricerca e Innovazione 2014–20*; ANR project SmartCloud ANR-23-CE25-0012.

References

[1] Tahereh Abbasi-khazaei and Mohammad Hossein Rezvani. 2022. Energy-aware and carbon-efficient VM placement optimization in cloud datacenters using evolutionary computing methods. *Soft Comput.* 26, 18 (2022), 9287–9322.

[2] Erika Ábrahám, Florian Corzilius, Einar Broch Johnsen, Gereon Kremer, and Jacopo Mauro. 2016. Zephyrus2: On the Fly Deployment Optimization Using SMT and CP Technologies. In *Dependable Software Engineering: Theories, Tools, and Applications - Second International Symposium, SETTA 2016, Beijing, China, November 9-11, 2016, Proceedings (Lecture Notes in Computer Science, Vol. 9984)*, Martin Fränzle, Deepak Kapur, and Naijun Zhan (Eds.). 229–245. doi:10.1007/978-3-319-47677-3_15

[3] Ehsan Ahvar et al. 2021. DECA: A Dynamic Energy Cost and Carbon Emission-Efficient Application Placement Method for Edge Clouds. *IEEE Access* 9 (2021), 70192–70213.

[4] Mohammad Aldossary and Hatem A. Alharbi. 2021. Towards a Green Approach for Minimizing Carbon Emissions in Fog-Cloud Architecture. *IEEE Access* 9 (2021), 131720–131732.

[5] Roberto Amadini, Simone Gazza, Jacopo Soldani, Monica Vitali, Antonio Brogi, Stefano Forti, Saverio Giallorenzo, Pierluigi Plebani, Francisco Ponce, and Gianluigi Zavattaro. 2024. Pick a Flavour: Towards Sustainable Deployment of Cloud-Edge Applications. In *Logic-Based Program Synthesis and Transformation - 34th International Symposium, LOPSTR 2024, Milan, Italy, September 9-10, 2024, Proceedings (Lecture Notes in Computer Science, Vol. 14919)*, Juliana Bowles and Harald Søndergaard (Eds.). Springer, 117–127. doi:10.1007/978-3-031-71294-4_7

[6] Hemant Kumar Apat et al. 2023. A comprehensive review on Internet of Things application placement in Fog computing environment. *Internet Things* 23 (2023), 100866.

[7] Lorenzo Bacchiani, Mario Bravetti, Saverio Giallorenzo, Maurizio Gabbrielli, Gianluigi Zavattaro, and Stefano Pio Zingaro. 2025. Proactive-reactive microservice architecture global scaling. *J. Syst. Softw.* 220 (2025), 112262. doi:10.1016/J.JSS.2024.112262

[8] Lorenzo Bacchiani, Mario Bravetti, Saverio Giallorenzo, Jacopo Mauro, Iacopo Talevi, and Gianluigi Zavattaro. 2021. Microservice Dynamic Architecture-Level Deployment Orchestration. In *Coordination Models and Languages*, Ferruccio Damiani and Ornela Dardha (Eds.). Springer International Publishing, Cham, 257–275.

[9] Albert-László Barabási and Réka Albert. 1999. Emergence of scaling in random networks. *science* 286, 5439 (1999), 509–512.

[10] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. 2016. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`.

[11] Mario Bravetti, Saverio Giallorenzo, Jacopo Mauro, Iacopo Talevi, and Gianluigi Zavattaro. 2019. Optimal and automated deployment for microservices. In *Fundamental Approaches to Software Engineering: 22nd International Conference, FASE 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6–11, 2019, Proceedings 22*. Springer International Publishing, 351–368.

[12] Antonio Brogi, Stefano Forti, and Ahmad Ibrahim. 2019. Predictive Analysis to Support Fog Application Deployment. In *Fog and Edge Computing*. Wiley, 191–221. doi:10.1002/9781119525080.ch9

[13] Brendan Burns, Joe Beda, Kelsey Hightower, and Lachlan Evenson. 2022. *Kubernetes up & running: Dive into the future of Infrastructure*. O'Reilly Media.

[14] Christian Schulte and Guido Tack. [n. d.]. Gecode: Generic Constraint Development Environment. http://www.gecode.org Last accessed in May, 2024..

[15] Geoffrey Chu et al. [n. d.]. Chuffed, a lazy clause generation solver. https://github.com/chuffed/chuffed Last accessed in May, 2024..

[16] Roberto Di Cosmo, Antoine Eiche, Jacopo Mauro, Stefano Zacchiroli, Gianluigi Zavattaro, and Jakub Zwolakowski. 2015. Automatic Deployment of Services in the Cloud with Aeolus Blender. In *Service-Oriented Computing - 13th International Conference, ICSOC 2015, Goa, India, November 16-19, 2015, Proceedings (Lecture Notes in Computer Science, Vol. 9435)*, Alistair Barros, Daniela Grigori, Nanjangud C. Narendra, and Hoa Khanh Dam (Eds.). Springer, 397–411. doi:10.1007/978-3-662-48616-0_28

[17] Roberto Di Cosmo, Michael Lienhardt, Ralf Treinen, Stefano Zacchiroli, Jakub Zwolakowski, Antoine Eiche, and Alexis Agahi. 2014. Automated synthesis and deployment of cloud applications. In *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*, Ivica Crnkovic, Marsha Chechik, and Paul Grünbacher (Eds.). ACM, 211–222. doi:10.1145/2642937.2642980

[18] Roberto Di Cosmo, Jacopo Mauro, Stefano Zacchiroli, and Gianluigi Zavattaro. 2014. Aeolus: A component model for the cloud. *Inf. Comput.* 239 (2014), 100–121. doi:10.1016/J.IC.2014.11.002

[19] Breno G. S. Costa et al. 2023. Orchestration in Fog Computing: A Comprehensive Survey. *ACM Comput. Surv.* 55, 2 (2023), 29:1–29:34.

[20] Bogdan David and Madalina Erascu. 2023. Benchmarking Optimization Solvers and Symmetry Breakers for the Automated Deployment of Component-based Applications in the Cloud. *arXiv preprint arXiv:2305.15231* (2023).

[21] Stijn de Gouw, Jacopo Mauro, and Gianluigi Zavattaro. 2019. On the modeling of optimal and automatized cloud application deployment. *J. Log. Algebraic Methods Program.* 107 (2019), 108–135. doi:10.1016/J.JLAMP.2019.06.001

[22] Shuiguang Deng, Zhengzhe Xiang, Javid Taheri, Mohammad Ali Khoshkholghi, Jianwei Yin, Albert Y Zomaya, and Schahram Dustdar. 2020. Optimal application deployment in resource constrained distributed edges. *IEEE transactions on mobile computing* 20, 5 (2020), 1907–1923.

[23] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch-Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. 2017. Microservices: Yesterday, Today, and Tomorrow. In *Present and Ulterior Software Engineering*, Manuel Mazzara and Bertrand Meyer (Eds.). Springer, 195–216. doi:10.1007/978-3-319-67425-4_12

[24] Mădălina Erașcu, Flavia Micota, and Daniela Zaharie. 2021. Scalable optimal deployment in the cloud of component-based applications using optimization modulo theory, mathematical programming and symmetry breaking. *Journal of Logical and Algebraic Methods in Programming* 121 (2021), 100664. doi:10.1016/j.jlamp.2021.100664

[25] Paul Erdős and Alfréd Rényi. 1959. On Random Graphs. I. *Publicationes Mathematicae Debrecen* 6, 3-4 (1959), 290–297. doi:10.5486/PMD.1959.6.3-4.12

[26] Sara Farzai, Mirsaeid Hosseini Shirvani, and Mohsen Rabbani. 2020. Multi-objective communication-aware optimization for virtual machine placement in cloud datacenters. *Sustainable Computing: Informatics and Systems* 28 (2020), 100374.

[27] Francescomaria Faticanti, Francesco De Pellegrini, Domenico Siracusa, Daniele Santoro, and Silvio Cretti. 2020. Throughput-Aware Partitioning and Placement of Applications in Fog Computing. *IEEE Trans. Netw. Serv. Manag.* 17, 4 (2020), 2436–2450. doi:10.1109/TNSM.2020.3023011

[28] Jeffrey Fischer, Rupak Majumdar, and Shahram Esmaeilsabzali. 2012. Engage: a deployment management system. *SIGPLAN Not.* 47, 6 (June 2012), 263–274. doi:10.1145/2345156.2254096

[29] Stefano Forti and Antonio Brogi. 2021. Declarative Osmotic Application Placement. In *CAiSE 2021 – Workshops (LNBIP, Vol. 423)*. Springer, 177–190.

[30] Stefano Forti and Antonio Brogi. 2022. Green Application Placement in the Cloud-IoT Continuum. In *PADL (LNCS, Vol. 13165)*. Springer, 208–217.

[31] Stefano Forti, Federica Paganelli, and Antonio Brogi. 2022. Probabilistic QoS-aware Placement of VNF Chains at the Edge. *Theory Pract. Log. Program.* 22, 1 (2022), 1–36. doi:10.1017/S1471068421000016

[32] Robert Fourer, David M. Gay, and Brian W. Kernighan. 1989. AMPL: A Mathematical Programming Language. In *Algorithms and Model Formulations in Mathematical Programming*, Stein W. Wallace (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 150–151.
[33] Marco Gaglianese et al. 2023. Green Orchestration of Cloud-Edge Applications: State of the Art and Open Challenges. In *SOSE*. 250–261.
[34] Simone Gazza and Roberto Amadini. 2024. Source code. https://github.com/simonegazza/mzn-test-suite
[35] Simone Gazza and Francisco Ponce. 2024. Parser. https://github.com/FREEDA-Project/Python-parser
[36] Saverio Giallorenzo, Jacopo Mauro, Martin Gyde Poulsen, and Filip Siroky. 2021. Virtualization Costs: Benchmarking Containers and Virtual Machines Against Bare-Metal. *SN Comput. Sci.* 2, 5 (2021), 404. doi:10.1007/s42979-021-00781-8
[37] Wedan Emmanuel Gnibga et al. 2023. Latency, Energy and Carbon Aware Collaborative Resource Allocation with Consolidation and QoS Degradation Strategies in Edge Computing. In *ICPADS*.
[38] Google AI. [n. d.]. OR-Tools. https://developers.google.com/optimization Last accessed in May, 2024..
[39] Juan Luis Herrera, Javier Berrocal, Stefano Forti, Antonio Brogi, and Juan Manuel Murillo. 2023. Continuous QoS-aware adaptation of Cloud-IoT application placements. *Computing* 105, 9 (2023), 2037–2059. doi:10.1007/s00607-023-01153-1
[40] Juan Luis Herrera, Jaime Galán-Jiménez, Javier Berrocal, and Juan Manuel Murillo. 2021. Optimizing the Response Time in SDN-Fog Environments for Time-Strict IoT Applications. *IEEE Internet Things J.* 8, 23 (2021), 17172–17185. doi:10.1109/JIOT.2021.3077992
[41] Mirsaeid Hosseini Shirvani and Yaser Ramzanpoor. 2023. Multi-objective QoS-aware optimization for deployment of IoT applications on cloud and fog computing infrastructure. *Neural Computing and Applications* 35, 26 (2023), 19581–19626.
[42] Menglan Hu, Hao Wang, Xiaohui Xu, Jianwen He, Yi Hu, Tianping Deng, and Kai Peng. 2024. Joint Optimization of Microservice Deployment and Routing in Edge via Multi-Objective Deep Reinforcement Learning. *IEEE Transactions on Network and Service Management* (2024).
[43] Qi Huangfu and JA Julian Hall. 2018. Parallelizing the dual revised simplex method. *Mathematical Programming Computation* 10, 1 (2018), 119–142.
[44] Mohammad Mainul Islam et al. 2023. Optimal placement of applications in the fog environment: A systematic literature review. *J. Parallel Distributed Comput.* 174 (2023), 46–69.
[45] Richard M. Karp. 1972. *Reducibility among Combinatorial Problems*. Springer US, Boston, MA, 85–103. doi:10.1007/978-1-4684-2001-2_9
[46] Torgeir Lebesbye, Jacopo Mauro, Gianluca Turin, and Ingrid Chieh Yu. 2021. Boreas – A Service Scheduler for Optimal Kubernetes Deployment. In *Service-Oriented Computing*, Hakim Hacid, Odej Kao, Massimo Mecella, Naouel Moha, and Hye-young Paik (Eds.). Springer International Publishing, Cham, 221–237.
[47] Vincenzo De Maio and Ivona Brandic. 2018. First Hop Mobile Offloading of DAG Computations. In *18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGRID 2018, Washington, DC, USA, May 1-4, 2018*. IEEE Computer Society, 83–92. doi:10.1109/CCGRID.2018.00023
[48] Jukka Manner. 2022. Black software — the energy unsustainability of software systems in the 21st century. *Oxford Open Energy* 2 (2022).
[49] Jacopo Massa, Stefano Forti, Patrizio Dazzi, and Antonio Brogi. 2023. Declarative and Linear Programming Approaches to Service Placement, Reconciled. In *IEEE CLOUD 2023*. 1–10. doi:10.1109/CLOUD60044.2023.00033
[50] Dirk Merkel. 2014. Docker: lightweight Linux containers for consistent development and deployment. *Linux J.* 2014, 239, Article 2 (March 2014).
[51] MiniZinc Teams. [n. d.]. MiniZinc Challenge 2024 - Rules. https://www.minizinc.org/challenge/2024/rules/#complete-scoring-procedure Last accessed in May, 2024..
[52] Nicholas Nethercote et al. 2007. MiniZinc: Towards a Standard CP Modelling Language. 4741 (2007), 529–543.
[53] Kai Peng, Liangyuan Wang, Jintao He, Chao Cai, and Menglan Hu. 2024. Joint optimization of service deployment and request routing for microservices in mobile edge computing. *IEEE Transactions on Services Computing* (2024).
[54] Vladimir Podolskiy, Maria Patrou, Panos Patros, Michael Gerndt, and Kenneth B. Kent. 2020. The weakest link: revealing and modeling the architectural patterns of microservice applications. In *CASCON '20: Proceedings of the 30th Annual International Conference on Computer Science and Software Engineering, Toronto, Ontario, Canada, November 10 - 13, 2020*, Lily Shaddick, Guy-Vincent Jourdan, Vio Onut, and Tinny Ng (Eds.). ACM, 113–122. doi:10.5555/3432601.3432616
[55] Amit M Potdar, Narayan D G, Shivaraj Kengond, and Mohammed Moin Mulla. 2020. Performance Evaluation of Docker Container and Virtual Machine. *Procedia Computer Science* 171 (2020), 1419–1428. doi:10.1016/j.procs.2020.04.152 Third International Conference on Computing and Network Communications (CoCoNet'19).
[56] Francisco Ponce Simone Gazza and Jacopo Soldani. 2024. YAML Model Specification. https://github.com/FREEDA-Project/YAML-model/tree/v0.2
[57] Jacopo Soldani et al. 2024. Towards Sustainable Deployment of Microservices over the Cloud-Edge Continuum, with FREEDA. In *Workshop on Flexible Resource and Application Management on the Edge (FRAME '24)*. ACM. doi:10.1145/3659994.3660311
[58] Peter J. Stuckey et al. 2014. The MiniZinc Challenge 2008-2013. *AI Magazine* 2 (2014), 55–60.
[59] Monica Vitali. 2022. Towards greener applications: enabling sustainable-aware cloud native applications design. In *International Conference on Advanced Information Systems Engineering*. Springer, 93–108.

[60] Monica Vitali, Paul Schmiedmayer, and Valentin Bootz. 2023. Enriching Cloud-native Applications with Sustainability Features. In *2023 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 21–31.
[61] Ye Xia, Xavier Etchevers, Loïc Letondeur, Thierry Coupaye, and Frédéric Desprez. 2018. Combining hardware nodes and software components ordering-based heuristics for optimizing the placement of distributed IoT applications in the fog. In *ACM Symposium on Applied Computing*. 751–760. doi:10.1145/3167132.3167215
[62] YAML Language Development Team. 2021. YAML Ain't Markup Language (YAML™) Version 1.2.2. https://yaml.org/spec/1.2.2/. Accessed: 2025-01.
[63] Zhanwei Yu et al. 2023. Less Carbon Footprint in Edge Computing by Joint Task Offloading and Energy Sharing. *IEEE Netw. Lett.* 5, 4 (2023), 245–249.
[64] Hailiang Zhao, Shuiguang Deng, Zijie Liu, Jianwei Yin, and Schahram Dustdar. 2020. Distributed redundant placement for microservice-based applications at the edge. *IEEE Transactions on Services Computing* 15, 3 (2020), 1732–1745.

Received xxx; revised xxx; accepted xxx