TPLP: Page 1–18.

doi:10.1017/S1471068425100148

Application Placement with Constraint Relaxation

DAMIANO AZZOLINI
University of Ferrara, Ferrara, Italy
(*e-mail:* damiano.azzolini@unife.it)

MARCO DUCA, FRANCESCO GALLO and ANTONIO IELO
University of Calabria, Calabria, Italy
(*e-mails:* marco.duca02@gmail.com, francescogallo0309@gmail.com, antonio.ielo@unical.it)

STEFANO FORTI
University of Pisa, Pisa, Italy
(*e-mail:* stefano.forti@unipi.it)

submitted 22 July 2025; revised 22 July 2025; accepted 27 July 2025

Abstract

Novel utility computing paradigms rely upon the deployment of multi-service applications to pervasive and highly distributed cloud-edge infrastructure resources. Deciding onto which computational nodes to place services in cloud-edge networks, as per their functional and non-functional constraints, can be formulated as a combinatorial optimisation problem. Most existing solutions in this space are not able to deal with *unsatisfiable* problem instances, nor preferences, i.e., requirements that DevOps may agree to relax to obtain a solution. In this article, we exploit Answer Set Programming optimisation capabilities to tackle this problem. Experimental results in simulated settings show that our approach is effective on lifelike networks and applications.

KEYWORDS: answer set programming, logic programming applications, cloud-edge computing, application management, distributed computing

1 Introduction

In the last decade, cloud-edge computing paradigms (e.g., fog, edge, mist computing) have attracted increasing attention from both academic and industrial research communities (Srirama (2024)). These paradigms extend the traditional cloud computing model by incorporating resources along a computing continuum, ultimately interconnecting Internet of Things (IoT) devices with cloud virtual machines through a hierarchy of intermediate layers spanning end-user devices, enriched infrastructure assets, and small-scale private data centres. Overall, they aim at offering computing, storage and networking as utilities by leveraging a continuum of pervasive, heterogeneous resources that enable low-latency processing and context-aware service delivery, especially targeting

latency-sensitive or bandwidth-intensive IoT applications, for example augmented reality, remote surgery or safety monitoring (Moreschini *et al.* (2022); Vetriveeran (2025)).

Deploying multi-service applications across cloud-edge resources comes with significant challenges due to the scale, heterogeneity and dynamic nature of resources, along with stringent Quality of Service (QoS) requirements of the applications to be deployed (Apat *et al.* (2025)). Such applications, composed of interacting services, must be placed to meet constraints such as latency, bandwidth, energy consumption, and locality of IoT devices. Traditional placement strategies often approach this as a constraint satisfaction or optimisation problem (Mahmud *et al.* (2020a)). In practice, certain combinations of constraints may be unsatisfiable, leading current methods to fail due to resource scarcity, too demanding application requirements, or both. Notably, the current literature on cloud-edge application placement does not address the case of requirements that should be *preferably*, but not necessarily, satisfied according to given priorities.

For instance, a DevOps may request that a real-time video analytics service should be deployed to a node capable of reaching a surveillance camera with suitable latency and bandwidth, while also imposing strict limits on the carbon intensity of the chosen node. If no node can satisfy all the constraints, existing approaches (Mahmud *et al.* (2020a)) would typically reject the deployment altogether. This highlights the need for a placement strategy capable of reasoning over conflicting constraints and identifying which ones can be relaxed with *minimal* impact, according to priorities defined by application DevOps.

In this context, we leverage Answer Set Programming (Brewka *et al.* (2011)) (ASP) for determining QoS-aware placements of multi-service applications within cloud-edge environments and propose a novel solution called FlexiPlace. The main novelty of our approach lies in its ability to manage unsatisfiable placement instances by selectively *relaxing* (dropping) constraints based on a priority hierarchy established by the application DevOps (the "domain experts"), allowing one to determine an eligible application deployment instead of failing. We assess our tool on a set of benchmarks and show that it is effective on realistic-sized infrastructures and applications.

The article is organised as follows. Section 2 discusses the background. Section 3 introduces the considered problem through a motivating scenario, which is encoded in ASP in Section 4. Section 5 presents the experimental evaluation, Section 6 surveys related work, and Section 7 concludes the paper.

2 Background

Answer Set Programming (ASP) is a popular declarative programming paradigm. Its compact and expressive language makes it a powerful tool for handling knowledge-intensive combinatorial problems, both in industry and academia (Erdem *et al.* (2016); Falkner *et al.* (2018); Gamblin *et al.* (2022); Baumeister *et al.* (2024); Azzolini *et al.* (2025)), also thanks to the availability of efficient reasoners (Leone *et al* (2006); Gebser *et al.* (2019)).

Syntax. A *term* is either a *variable*, a *constant,* or a *function symbol*, where variables start with uppercase letters and constants start with lowercase letters or are numbers. A function term is an expression of the form $f(t_1, \ldots, t_n)$ where f is its name and t_i are terms. An *atom* is an expression of the form $p(t_1, \ldots, t_n)$ where p is a predicate of

https://doi.org/10.1017/S1471068425100148 Published online by Cambridge University Press

arity n and $t_1, \ldots, t_n$ are terms; it is *ground* if all its terms are constants. A *literal* is either an atom a or its negation $not\, a$, where *not* denotes the negation as failure. A literal is said to be *negative* if it is of the form $not\, a$, otherwise it is *positive*. For a literal l, $\bar{l}$ denotes the complement of l. More precisely, $\bar{l} = a$ if $l = not\, a$, otherwise $\bar{l} = not\, a$. A *normal rule* is an expression of the form $h \leftarrow b_1, \ldots, b_n$ where h is an atom called *head*. When $n \geq 0$, $b_1, \ldots, b_n$ is a conjunction of literals called *body*. A normal rule is said to be a *constraint* if its head is omitted, while it is said to be a *fact* if $n = 0$. A *programme* is a finite set of normal rules. We will also use choice rules (Niemelä *et al.* (1999)). A *choice element* is of the form $h : l_1, \ldots, l_k$, where h is an atom, and $l_1, \ldots, l_k$ is a conjunction of literals. A *choice rule* is an expression of the form $\{e_1; \ldots; e_m\} \leftarrow b_1, \ldots, b_n$. We also consider *aggregate* atoms in the body of rules (Alviano and Faber (2018)) of the form $\#sum\{\epsilon_0; \ldots; \epsilon_n\} > k$ where k is called *guard* and can be a constant or a variable and $\epsilon_0, \ldots, \epsilon_n$ is such that each ϵ_i has the form $t_1, \ldots, t_n : F$ and each t_i is a term whose variables appear in the conjunction of literals F.

Semantics. Given a programme P and $r \in P$, $ground(r)$ is the set of ground instantiations of r obtained by replacing variables in r with constants in P. For aggregates, a variable is called *local* if it appears only in the considered aggregate; *global* otherwise. The grounding of a rule with aggregates first requires replacing global variables and then replacing local variables appearing in aggregates with ground terms. We denote with $ground(P)$ the union of ground instantiations of rules in P. An aggregate is true in an interpretation I (i.e., a set of atoms) if the evaluation of the aggregate function under I satisfies the guards. We refer the reader to Calimeri *et al.* (2020) for a more in-depth treatment of aggregates. Given a programme P, an *interpretation* I is an *answer set* (also called stable model) of P iff (i) I is a model, that is for each rule $r \in ground(\Pi)$ either the head of r is true w.r.t. I or the body of r is false w.r.t I; and (ii) I is a minimal model of its GL-reduct (Gelfond and Lifschitz (1991)). If P has no answer sets, it is called *unsatisfiable*.

Optimisation. Weak constraints (Buccafurri *et al.* (2000)) are expressions of the form $:\sim l_1, \ldots, l_m.[w@p, t_1, \ldots, t_n]$ where $l_1, \ldots, l_m$ are literals, $w \in \mathbb{N}$ is the cost, $p \in \mathbb{N}$ is the priority, and $t_1, \ldots, t_n$ is a tuple of terms. Such rules associate each answer set with a *cost* with a *priority level*, which can be intuitively understood as an objective function, to be optimised in order of priority. These enable one to tackle optimisation problems in ASP (Alviano *et al.* (2020)). An answer set with costs $c_0, c_1, \ldots c_k$ has a "lower cost" than an answer set with costs $c'_0, c'_1, \ldots, c'_k$ if there exist i such that $c_i < c'_i$ and $c_j = c'_j$ for all $j < i$. An answer set is *optimal* if there does not exist an answer set with a lower cost.

3 Motivating scenario: the application placement problem

In this section, we illustrate the considered problem by means of a simple, yet complete, motivating example adapted from the literature (Forti (2022)). The depicted scenario epitomises a broader class of placement problems in which functional (e.g., hardware, IoT) and non-functional requirements related to sustainability (e.g., energy efficiency, carbon intensity), performance (e.g., latency, bandwidth), and reliability (e.g., availability, security) must be satisfied simultaneously, despite being often conflicting and constrained by

https://doi.org/10.1017/S1471068425100148 Published online by Cambridge University Press

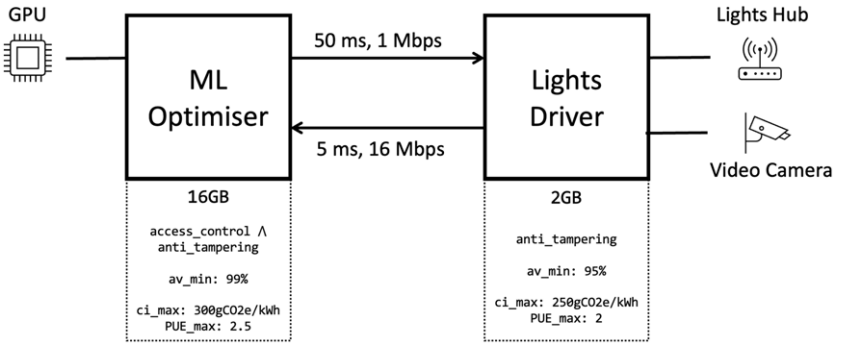

Fig. 1. Example application.

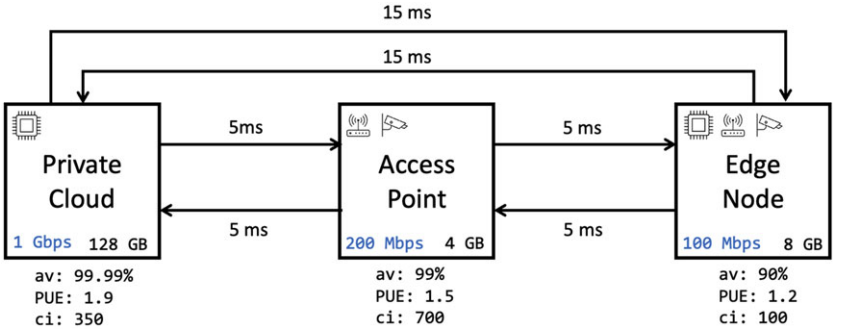

Fig. 2. Example infrastructure.

limited resources. Typical deployments involve hundreds of services and nodes, leading to a combinatorial explosion of potential candidate solutions (Smolka and Mann (2022)) (e.g., mapping 30 services on 100 nodes yields up to $100^{30} = 10^{60}$ candidates to check).

The application in Figure 1 manages street lighting using machine learning (ML) and includes two services: the ML Optimiser, which processes video streams to determine optimal lighting strategies, and the Lights Driver, which controls the street lights. The ML Optimiser requires a GPU co-processor to train models that update the driver's control rules while the Lights Driver interfaces with both a lighting hub and a video camera, which monitors ambient conditions and streams footage to the optimiser. Each service has functional and non-functional requirements. For instance, the ML Optimiser requires 16 GB RAM, access control and anti-tampering mechanisms, minimum node availability (`av_min`) of 99%, carbon intensity of the node energy mix (`ci_max`) below 300 gCO_2-eq/kWh, and power usage effectiveness[1] (`PUE_max`) under 2.5. Additionally, communication constraints specify a maximum latency of 50 ms and a minimum bandwidth of 1 Mbps from the ML Optimiser to the Lights Driver, and 5 ms and 16 Mbps in the reverse direction.

Figure 2 sketches the target infrastructure for the described application. It consists of three interconnected computing nodes – Private Cloud, Access Point, and Edge Node – with

[1] PUE is a standard computing efficiency metric defined as the ratio of total energy consumption of an IT system to the energy used by computing equipment alone. A value of 1.0 indicates ideal efficiency.

https://doi.org/10.1017/S1471068425100148 Published online by Cambridge University Press

a different availability of resources. For instance, the Edge Node is located closest to end devices, being directly connected to the lights hub and the video camera, and is equipped with 8 GB of RAM, a GPU, and 100 Mbps mobile connectivity. It has 90% availability, a PUE of 1.2, and is powered up through an energy mix with a carbon intensity of 100 gCO_2-eq/kWh. While the Private Cloud is assumed to feature all security mechanisms, the other two nodes are only equipped with an anti-tampering system to mitigate the damage in case of physical access to deployment resources. Network communication latencies are symmetric and range from 5 ms between adjacent nodes to 15 ms between the Private Cloud and the Edge Node.

Determining an eligible placement of the application of Figure 1 to the resources of Figure 2 requires mapping each service to a node that satisfies *all* its functional and non-functional requirements, without exceeding the node's hardware and bandwidth capacity, which is an NP-hard problem (Brogi and Forti (2017)). In the described scenario, there is no solution placement that can satisfy all application requirements in the target infrastructure. To address this, however, the DevOps team in charge of application deployment is willing to relax certain non-functional requirements (i.e., *soft*) based on predefined priorities, where higher values indicate greater importance and less flexibility. Latency and bandwidth constraints have the highest priority (10) and are thus the least negotiable. Availability follows with priority 2, indicating moderate flexibility. Conversely, PUE and carbon intensity have the lowest priority (1). Security, hardware and IoT requirements remain non-negotiable (i.e., *hard*) and must always be strictly enforced.

Hereinafter, we show how FlexiPlace leverages ASP and accounts for constraint prioritisation to relax a minimal set of requirements to compute feasible placements with the least impact on the original constraints. That is, we aim at answering the following question: *How can we determine a constraint-relaxed placement of a cloud-edge application that is both feasible and minimally deviates from our original deployment intent?*

4 ASP encoding

The core idea of our encoding is to match services to nodes in the infrastructure, discarding candidate solutions that do not satisfy *application requirements* over infrastructure *capabilities*. As we will formalise in the following, we represent an infrastructure as a graph extended with *attributes*. Analogously, applications are represented as graphs and their requirements consist of simple arithmetic relationships between a (constant) threshold value for an attribute that a given service "requires" and the value of that specific attribute over the *node where a service is placed*. Rather than considering a fixed set of attributes for the nodes (and application requirements), our solution provides a way (by means of input facts) to *define* the attributes of each node (and the corresponding requirements), that can be thus customised by users.

We first describe how to encode an input instance (infrastructure & its capabilities, application & its requirements) into a set of ASP facts, then how to model the application placement problem in ASP ("base encoding"), and finally how to address its relaxed version through ASP optimisation, using the clingo (Gebser *et al.* (2019)) input language.

https://doi.org/10.1017/S1471068425100148 Published online by Cambridge University Press

4.1 Reification of infrastructures & applications

Modelling the infrastructure. We model a target deployment infrastructure by means of predicates $node/1$ and $link/2$. We use an atom $node(x)$ for each vertex x in the network, and an atom $link(x, y)$ for each edge that connects nodes x and y. Informally, both node attributes and link attributes are modelled as "key-value pairs," by means of predicates $node\_attr/3$ and $link\_attr/4$. Atoms $node\_attr(x, k, v)$ mean that attribute k has value v on node x, and $link\_attr(x, y, k, v)$ mean that attribute k on edge (x, y) has value v. This representation accommodates several kinds of infrastructure and application properties.

Example 1 (Infrastructure).
The node Private Cloud *of Figure 2 is encoded via the following set of facts:*

```
node("prvt_cloud").
node_attr("prvt_cloud","access_control",true). node_attr("prvt_cloud","anti_tampering",true).
node_attr("prvt_cloud","availability",9999).   node_attr("prvt_cloud","bandwidth_in",1000).
node_attr("prvt_cloud","bandwidth_out",1000).  node_attr("prvt_cloud","carbon_intensity",350).
node_attr("prvt_cloud","gpu",true).            node_attr("prvt_cloud","pue",19).
node_attr("prvt_cloud","ram_gb",128).
```

Similarly, the link that connects such node to the Edge Node *is denoted by the fact:*

```
link_attr("prvt_cloud", "edge_node", "latency", 15).
```

Modelling the application. We use the predicates $service/1$ and $dependency/2$, with an analogous meaning to $node/1$ and $link/2$, to describe the application to be placed. That is, $service(s)$ denotes that s is a unique identifier for a service, and $dependency(s, t)$ that service s "depends on" service t. The counterpart of infrastructure attributes are service *requirements*, which intuitively act as "constraints"[2] to forbid deployments. Intuitively, as services are deployed onto nodes and have pairwise dependencies, we can label each service with properties *its matchee* must abide. Thus, while infrastructure attributes refer to its vertices and edges, service requirements will refer to *the node the service is deployed onto (during stable model search).*

A *(simple) requirement* is a statement about infrastructure properties a node should possess in order to host a service. Here, we focus on requirement expressions that consist of comparisons between attributes' value and constants. That is, expressions of the form $r \circ t$ where r is an attribute, $\circ \in \{<, >, \leq, \geq, =, \neq\}$ or $reserve(r, t)$. Intuitively, these respectively mean that the value (on a node) of an attribute r should compare in a specific way against a threshold value t. The requirement expression $reserve(r, t)$ expresses that (i) attribute r should be understood as consumable resources (ii) a given service requires t units of r on the node to which it is deployed. Application deployments often require that each service has access to a dedicated amount of computational resources (e.g., RAM or bandwidth). Whenever multiple services are placed onto the same node or link asset, it must be ensured that the consumable resources are enough to host all services.

We encode requirement expressions by means of function terms such as $lt(r, v)$ ("less-than"), $gt(r, v)$ ("greater-than"), $eq(r, v)$ ("equals-to"), in addition to the aforementioned

[2] In the rest of the section, we purposefully avoid using the word *deployment constraints* as it is semantically overloaded with the notion of constraints in ASP.

https://doi.org/10.1017/S1471068425100148 Published online by Cambridge University Press

$reserve(r, v)$. Given a requirement expression, we say that it holds on a given service using the atoms $hreq/2$ and $sreq/2$, which stand respectively for **h**ard **req**uirement and **s**oft **req**uirement. The atom $hreq(s, e)$ denotes that the service s can be deployed onto a node if and only if the node *satisfies* the requirement e. The atom $sreq(s, e)$ denotes that the service s should be deployed onto a node that *preferably* satisfies the requirement e. Similarly, the atoms $sreq((x, y), e)$ and $hreq((x, y), e)$ state that the requirement e must be satisfied by the link between the nodes that host the services x and y.

Example 2 (Application requirements).
The following encoding denotes the requirements of service ML Optimiser *of Figure 1*:

```
hreq("ml_opt",eq("access_control",true)).  hreq("ml_opt",eq("anti_tampering",true)).
sreq("ml_opt",gte("availability",99),2).   hreq("ml_opt",reserve("bandwidth_in",16)).
hreq("ml_opt",reserve("bandwidth_out",1)). sreq("ml_opt",lte("carbon_intensity",300)).
hreq("ml_opt",eq("gpu",true)).             sreq("ml_opt",lte("pue",25)).
hreq("ml_opt",reserve("ram_gb",16)).
```

Akin to infrastructure links, dependencies between services are denoted as:

```
sreq(("ml_opt", "lights_driver"), lte("latency", 50)).
sreq(("lights_driver", "ml_opt"), lte("latency", 5)).
```

4.2 Encoding application deployment

We now present the "base" encoding that solves the deployment problem. It is based on a guess-and-check procedure, where choice rules guess a candidate assignment of services to infrastructure nodes and constraints prune assignments that violate requirements. We denote such a programme Π_{deploy}. For now, *we assume no distinction between hard requirements (hreq/2) and soft requirements (sreq/2)*. Given an application network A and an infrastructure R, the answer sets of $\Pi_{deploy} \cup [R] \cup [A]$ can be mapped back to assignments that solve our problem. A solution placement can be decoded by projecting answer sets onto the $deploy/2$ predicate.

4.3 Relaxing soft requirements on application deployment

Distinguishing between hard and soft requirements naturally corresponds to an ASP optimisation task. The idea is to *abduce over possible constraints* to remove atoms matching $sreq/2$ by means of choice rules; this disables the corresponding constraint in the logic programme. To do so, (i) we replace line 9 in the base encoding (Figure 3) with the rules in Figure 4 and (ii) if we wish to weight (e.g., assign a preference score) to soft requirements to remove, we introduce atoms $violation\_cost(S, E, (C, L))$ to denote that we will pay a cost of C at level L if we renounce the soft requirement $sreq(S, E)$.

Optimal answer sets correspond to optimal solutions of the deployment problem, where atoms $lift(S, E)$ denote that we renounce the requirement E on the deployment of service S. If this logic programme is unsatisfiable, it means that even removing all $sreq$s, this *would not be sufficient to ensure existence of a deployment*. That is, (at least one of) the reason(s) for the inconsistency lies in $hreq$s alone, and further analysis would be required. The logic programme is satisfiable if there exists an assignment that perfectly fits all the requirements. As we are interested in detecting soft requirements to

https://doi.org/10.1017/S1471068425100148 Published online by Cambridge University Press

```
resource(R) :- node_attr(_,R,_).
% Deploy each service onto one node in the infrastructure.
{ deploy(S,X): node(X) } = 1 :- service(S).
% hreq: hard requirements (can't be relaxed)
% sreq: soft requirements (can be relaxed)
req(S,E) :- hreq(S,E).
req(S,E) :- sreq(S,E).
% Cumulative usage of resource
shared_resource(R) :- req(_,reserve(R,_)).
% Sum of all quantities Q of resource R reserved by services deployed in X
% must be below availability of R on Q
:- node_attr(X,R,T), shared_resource(R), #sum{Q,S: deploy(S,X), req(S,reserve(R,Q))} > T.
:- req(S,reserve(R,Q)), deploy(S,X), node_attr(X,R,V), V < Q.
% Enforcing requirement expressions (nodes).
:- req(S,eq(R,V)),  deploy(S,X), not node_attr(X,R,V).
:- req(S,neq(R,V)), deploy(S,X), node_attr(X,R,V).
:- req(S,lt(R,T)),  deploy(S,X), node_attr(X,R,V), V >= T.
:- req(S,gt(R,T)),  deploy(S,X), node_attr(X,R,V), V <= T.
:- req(S,gte(R,T)), deploy(S,X), node_attr(X,R,V), V < T.
:- req(S,lte(R,T)), deploy(S,X), node_attr(X,R,V), V > T.
% Enforcing requirement expressions (edges).
:- req((S1,S2),eq(R,V)),  deploy(S1,X), deploy(S2,Y),  not link_attr(X,Y,R,V).
:- req((S1,S2),neq(R,V)), deploy(S1,X), deploy(S2,Y),  link_attr(X,Y,R,V).
:- req((S1,S2),lt(R,T)),  deploy(S1,X), deploy(S2,Y),  link_attr(X,Y,R,V), V >= T.
:- req((S1,S2),gt(R,T)),  deploy(S1,X), deploy(S2,Y),  link_attr(X,Y,R,V), V <= T.
:- req((S1,S2),gte(R,T)), deploy(S1,X), deploy(S2,Y),  link_attr(X,Y,R,V), V < T.
:- req((S1,S2),lte(R,T)), deploy(S1,X), deploy(S2,Y),  link_attr(X,Y,R,V), V > T.
```

Fig. 3. FlexiPlace main encoding.

```
{ req(S,E) } :- sreq(S,E,_).
lift(S,E) :- sreq(S,E), not req(S,E).
:~ violation_cost(S,E,(C,L)), lift(S,E). [C@L,S,E]
```

Fig. 4. Additions to the main encoding of Figure 3 to address the relaxed problem.

relax rather than finding deployments, we consider projection of answer sets on the *lift*/2 predicate. From the ASP modelling point of view, it would have been equivalent (in terms of optimal solutions) to directly express soft requirements as weak constraints. However, our design choices have several practical advantages: (*i*) the *lift*/*2* predicate enables to easily inspect answer sets and retrieve *which* requirements have been relaxed to achieve the solution, (*ii*) it is possible to control by means of facts (i.e., those matching *sreq*/2 and *hreq*/2) which requirements are mandatory and which ones can be relaxed, and (*iii*) atoms *lift*/*2* could be naturally used as *objective atoms* to compute minimal unsatisfiable subprograms for explainability purposes (Alviano *et al.* (2023)).

Example 3.
Consider again the motivating scenario of Section 3. As discussed above, there is no eligible placement that meets all the requirements for the application of Figure 1 to the infrastructure of Figure 2. For instance, the only node that can support the execution of the ML Optimiser *(i.e., the* Private Cloud *node) features a carbon intensity of 350 gCO_2-eq which exceeds the required 300 gCO_2-eq. Note that our model relies on facts like*

```
violation_cost(("ml_optimiser","lights_driver"),lte("latency",50),(10,1)).
```

https://doi.org/10.1017/S1471068425100148 Published online by Cambridge University Press

to set the cost (i.e. priority) for relaxing soft constraints. Running the encoding of Figures 3 and 4 over the input denoting our motivating scenario returns an eligible placement that suggests deploying ML Optimiser *to* Private Cloud *and* Lights Driver *to* Access Point, *and is obtained by relaxing constraints on carbon intensity for both services. Such a solution is optimal, as it only relaxes two of the lowest-priority constraints, as indicated by the DevOps team in charge of managing the application.*

5 Experiments

We perform a set of experiments to assess the effectiveness of our approach in computing cost-optimal relaxed deployments. We first analyse the behaviour of available ASP systems on our encoding. Then, we conducted a more in-depth analysis to investigate the trade-off between model-guided (Gebser *et al* (2011)) and core-guided (Andres *et al.* (2012)) optimisation algorithms for our application scenario.

The experiments[3] were run on a server with Intel(R) Xeon(R) CPU E7-8880 v4 @ 2.20 GHz CPU, equipped with 500 GB RAM, with a timeout of 180 s using GNU Parallel and executing at most 16 jobs in parallel. A timeout in our setting refers to not being able to find an *optimal model* or proving unsatisfiability within 180 s.

Data. We provide an instance generator for the problem, following standard practice in literature (Gupta *et al.* (2017); Forti *et al.* (2022); Ghobaei-Arani and Shahidinejad (2022)). We focus on the infrastructure attributes in Table 1 and define a set of realistic templates for infrastructure nodes and for application services. Infrastructure graphs and application graphs are obtained by sampling from Barabási-Albert (Barabási and Albert (1999)) and Erdős-Renyi (Erdős and Rényi (1959)) topologies, respectively. On one hand, the Barabási-Albert model captures the scale-free property of real-world ICT networks, where node degree distribution usually follows a power-law (Newman (2010)). This reflects the heterogeneity and hierarchy typical of cloud-edge infrastructures, where a small number of nodes act as high-bandwidth hubs and others as resource-constrained peripheral nodes. On the other hand, the Erdős-Renyi model neutrally approximates application topologies, where dependencies between services are established with uniform probability (Newman (2010)). This reflects the loosely coupled and stochastic nature of microservice-based applications, where interactions do not follow any hierarchical patterns (Soldani *et al.* (2018); Velepucha and Flores (2023)). Each node is assigned (uniformly at random) a "configuration" from a finite set. In our case, the only link attribute is latency, which we model as a random integer between 10 and 50. Nodes that are not directly connected by an edge are assigned an edge with a latency equal to the sum of the latencies in the shortest path between the two nodes. As an example, if a generated graph contains the edges (a, b) and (b, c), we introduce the edge (a, c) with latency $\ell(a, b) + \ell(b, c)$, where $\ell(x, y)$ is the latency on edge (x, y). Following these procedures, we generated 10 infrastructures of size $\{50, 100, 150, \ldots, 500\}$, and 6 applications of size $\{5, 10, 15, \ldots, 30\}$. For each combination, we generated 100 input pairs (i.e., application and infrastructure). This yields a total of $10 \cdot 6 \cdot 100 = 6 \cdot 10^3$ instances. We denote by $I(n, k)$ the set of instances considering the deployments of an application of size k over

[3] All experimental code and data are available at https://github.com/ainnoot/flexiplace.

https://doi.org/10.1017/S1471068425100148 Published online by Cambridge University Press

Table 1. *Considered node properties. The table does not contain latency since it is a link property. The "Shared" column denotes whether the property is considered shared among all services hosted on the considered nodes. The "Constr" column reports the constraint instantiated for the attribute during the generation process, for each service*

Attribute	Type	Shared	Constr	Attribute	Type	Shared	Constr
Access Control	Bool	No	Equals	CPU	Int	Yes	Reserve
Anti-tampering	Bool	No	Equals	Encryption	Bool	No	Equals
Availability	Int	No	At Least	GPU	Bool	No	Equals
Bandwidth (In)	Int	Yes	Reserve	Latency	Int	No	At Most
Bandwidth (Out)	Int	Yes	Reserve	PUE	Int	No	At Most
Carbon Intensity	Int	No	At Most	RAM	Int	Yes	Reserve
Cost	Int	No	At Most	Storage	Int	Yes	Reserve

infrastructures of size n. We also use the notation $I(\cdot, \{k_0, k_1, \ldots\})$ to denote "all problem instances that deal with applications of size k_0, k_1,, ..." and $I(\{n_0, n_1, \ldots\}, \cdot)$ to denote "all problem instances that deal with infrastructures of size n_0, n_1, ...". Lastly, each of the considered applications consists of constraints over *all* infrastructure properties defined in Table 1. Note that we use a weight of 1 for all constraints that can be relaxed, with a single priority level, but the encoding we provide is more general.

5.1 Solver selection

We consider four ASP systems, obtained by pairing up the ASP solvers `clasp` (Gebser *et al.* (2007a)) and `wasp` (Alviano *et al.* (2015)) with the ASP grounders I-DLV (Calimeri *et al.* (2017)) and `gringo` (Gebser *et al.* (2007b)). Note that the `gringo`+clasp and `IDLV`+`wasp` combinations are, essentially, the combinations adopted in the `clingo` (Gebser *et al.* (2019)) and `DLV` (Leone *et al* (2006)) solvers. We refer to each system as `gringo`+`wasp`, `gringo`+`clasp`, `IDLV`+`wasp`, `IDLV`+`clasp`. We execute the systems using both a *model-guided* (BB) (Gebser *et al* (2011)) and a *unsatisfiable core-guided* (USC) (Andres *et al.* (2012)) algorithm, which typically yield complementary performances (Alviano *et al.* (2020)). In brief, model-guided algorithms attempt to iteratively improve lower bound solutions, *à la* branch & bound, while unsatisfiable core-guided algorithms try to treat all weak constraints as standard (strong) constraints, using unsatisfiable cores found within the optimisation routine to shrink the search space. We consider the `bb` and `oll` algorithms for `clasp`, and the `basic` and `one` algorithms for `wasp`, which are the default for the model-guided and core-guided algorithms in these solvers, respectively. This yields a total of 8 configurations that we run over the 10% of the total instances, selecting 60 instances for each network size, for a total of 600 instances.

Figure 5 shows that, overall, `clasp`-based configurations outperform all `wasp`-based configurations in terms of execution times. In particular, the core-guided configuration of the `gringo`+`clasp` system (i.e., `clingo` with default parameters) essentially overlaps with the *virtual best solver*. Recall that the virtual best solver is a fictitious system that is assumed to perform (instance-wise) as the best among the available solvers. The scatter plots provide an instance-wise comparison of the systems. We can observe in Figure 6

https://doi.org/10.1017/S1471068425100148 Published online by Cambridge University Press

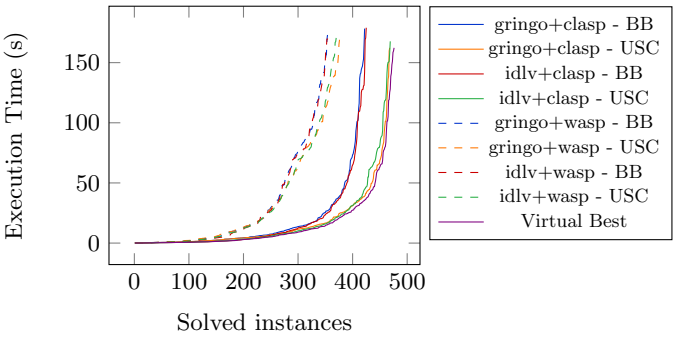

Fig. 5. Solvers comparison.

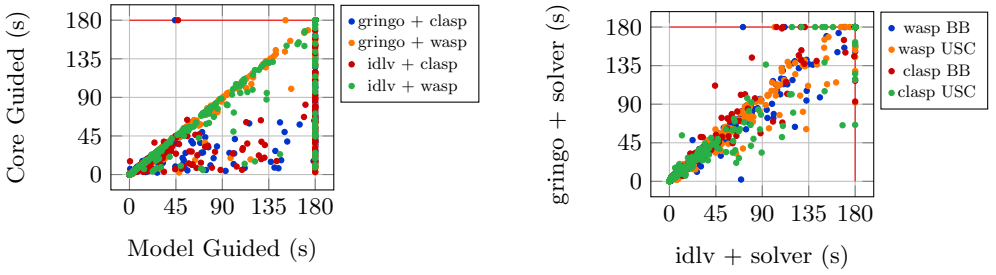

Fig. 6. Left: a point (x, y) denotes that a given problem instance is solved in x seconds using the model-guided and in y seconds using the core-guided algorithm. Right: a point (x, y) denotes that a given problem instance is solved in x seconds using the `IDLV` and in y seconds using `gringo` grounder.

(left) that the default core-guided algorithm outperforms the default model-guided algorithm across all systems. Moreover, Figure 6 (right) confirms that the grounder plays a less important role in our problem, with points distributed along the bisector.

5.2 Assessment of FlexiPlace

From the previous set of experiments the `gringo+clasp` system (i.e.,, `clingo`) obtained the best overall performance among the tested configurations. Thus, we focus on that system and perform a more in-depth assessment of our approach on *all* the generated instances $(6 \cdot 10^3)$. We start by discussing the easy instances and then continue with an in-depth analysis of the results in the harder instances. Memory-wise, in all settings, we do not report significant memory usage.

Easy instances. Applications of size up to 15 yield solvable instances for both optimisation techniques. Figure 7 reports average runtime (up to the first optimal solution) over instances $I(\cdot, \{5, 10, 15\})$, where we can observe indeed an exponential-like effect of the application size w.r.t. infrastructure size on the overall runtime regardless of the solving algorithm. Here, BB and USC have similar performances. Overall, this is already sufficient to show applicability of our technique on non-trivial deployments, over realistic-sized infrastructures. We continue our analysis, focusing on deployments with 20, 25 and 30 applications, being these more challenging.

https://doi.org/10.1017/S1471068425100148 Published online by Cambridge University Press

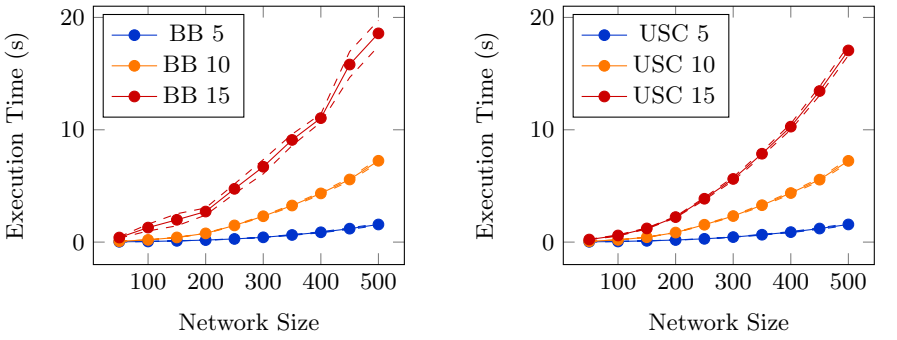

Fig. 7. Mean (solid) and standard error (dashed) on execution time to find the first optimal solution over $I(\cdot, \{5, 10, 15\})$ using BB (left) and USC (right) algorithms.

	# services					
	20		25		30	
size	BB	USC	BB	USC	BB	USC
50	28	3	35	38	28	37
100	39	6	82	50	99	94
150	32	4	83	49	98	94
200	18	0	68	28	96	89
250	24	1	75	28	99	95
300	21	3	62	37	96	89
350	22	1	68	40	96	91
400	19	5	76	46	100	95
450	25	14	64	44	93	89
500	21	6	65	43	93	83

Fig. 8. Number of timeouts per application size and network size.

Harder instances. Instances $I(\cdot, \{20, 25, 30\})$ yield more interesting behaviour and deserve further analysis. First, we observe in Table 8 that several instances hit the time-limit, with both optimisation algorithms. Figure 9 reports the overall performance of the two optimisation algorithms over these instances, in terms of a cactus plot. Overall, we can observe that these instances are better suited to be solved with USC techniques, as it is able to solve many more instances to optimality. Instance-wise, the scatter plot in Figure 10 confirms the result, usually with USC outperforming BB. However, USC also accrues more and more time-limits as the application size increases.

Temporal Distribution of Sub-optimal Answer Sets in BB. In practical scenarios, obtaining sub-optimal solutions in a fast way might still be useful. Thus, one might be interested in investigating whether sub-optimal solutions are obtained at all, whenever optimal solutions are unavailable. Figure 11 provides a plot on how non-optimal answer sets are found and distributed within allowed runtime when using the model-guided BB algorithm. We observe that overall *some* solutions (for all instances) are found within the first minute, then answer sets become more sparse, up to timeout – that typically occurs whenever the solver "hits an optimal model," but is not able yet to certify it as optimal (e.g., proving non-existence of a model with lesser cost).

https://doi.org/10.1017/S1471068425100148 Published online by Cambridge University Press

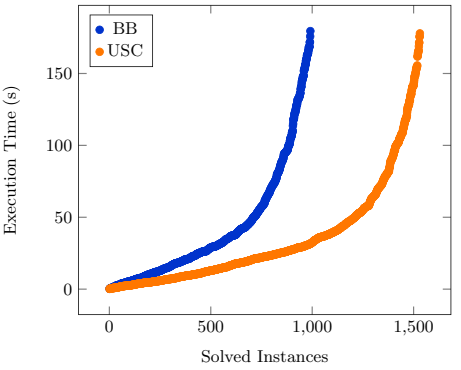

Fig. 9. Cumulative runtime of BB and USC algorithms over all instances.

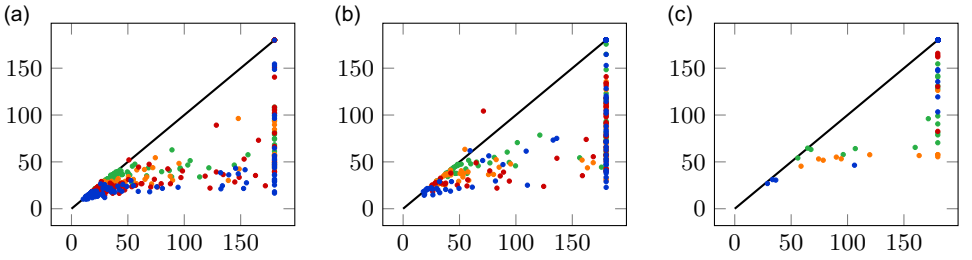

Fig. 10. Execution time over instances of size 350 (blue), 400 (red), 450 (orange), and 500 (green) in (a) $I(\{350, 400, 450, 500\}, 20)$, (b) $I(\{350, 400, 450, 500\}, 25)$, and (c) $I(\{350, 400, 450, 500\}, 30)$. A point (x, y) denotes that the USC algorithm solves an instance in y seconds, while BB solves it in x seconds.

Overall, USC and BB performance are comparable over 5–10–15 instances, regardless of network size, while USC is generally preferable for "harder" instances in the 20–25–30 range. However, BB is a way to obtain sub-optimal solutions quickly, whereas USC would time-out. We remark that, in a real-world setting, it would be totally feasible to run both approaches in parallel, so as to pick the first (optimal) solution found by either approach.

6 Related work

As aforementioned, the problem of deciding how to place application services to cloud-edge nodes in a QoS- and context-aware manner has been thoroughly studied. Here, we focus on the most closely related work, and we refer the readers to recent surveys by Pallewatta *et al.* (2023); Apat *et al.* (2025), and Aït-Salaht *et al.* (2021) for further details.

Many solutions exist that rely on different techniques to determine application placements that meet functional and non-functional requirements. Among these, most of the approaches relied on informed (heuristic) search (Brogi and Forti (2017); Gupta *et al.*

https://doi.org/10.1017/S1471068425100148 Published online by Cambridge University Press

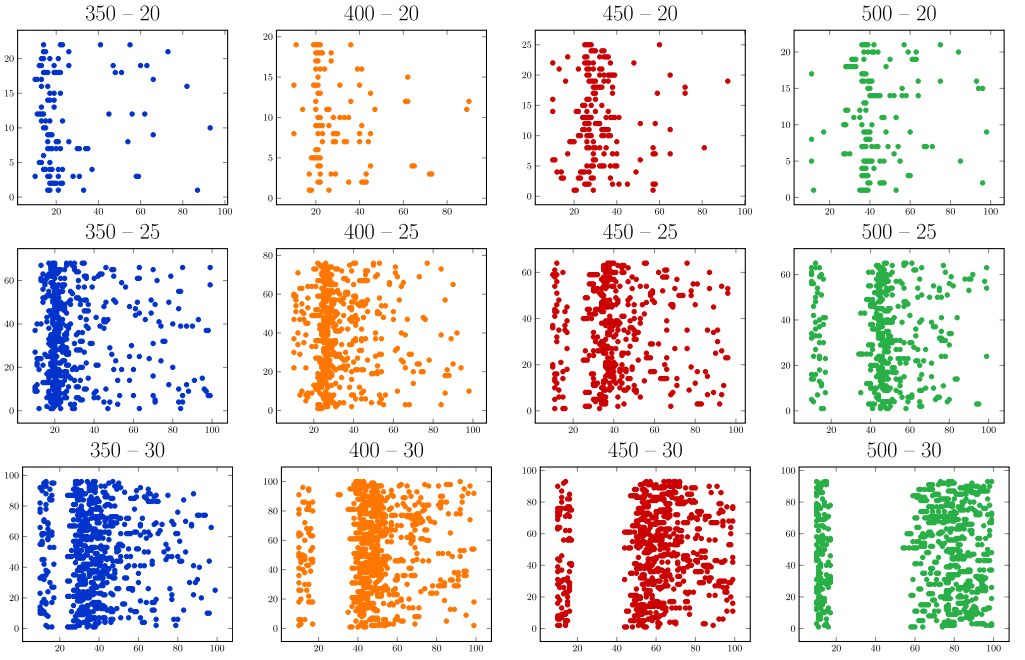

Fig. 11. Temporal distribution of answer sets using the BB algorithm. Each cell reports, in a dotted chart, data about $I(n, k)$ (with label n—k). A point (x, y) denotes that invoking the ASP solver on the y-th instance of $I(n, k)$ yields a model at time x.

(2017)), mathematical programming (Skarlat *et al.* (2017); Mahmud *et al* (2020b)), bio-inspired meta-heuristics (Ghobaei-Arani and Shahidinejad (2022)), and deep learning solutions (Goudarzi *et al.* (2023)). Different solutions aims at optimising one (or more) aspect(s) of application placements, for example operational costs, latency, energy consumption, and resource usage. Logic programming solutions, mainly written in Prologue, have recently been proposed to tackle the application placement problem, with a focus on aspects such as data locality (Massa *et al* (2022)), security and trust requirements (Forti *et al.* (2020)), environmental impact (Forti and Brogi (2022)), or high-level network intent satisfaction (Massa *et al.* (2024)). Notably, by classifying intent properties as either hard or soft, the latter approach supports recommending changes to original intents aimed at resolving emerging conflicts.

Forti *et al.* (2022) relied on continuous reasoning mechanisms to enable incremental updates of solution placements in response to changes in application requirements or infrastructure capabilities, rather than recomputing solutions from scratch. On a similar, yet complementary line, Azzolini *et al.* (2025) proposed a solution combining ASP optimisation and Prologue-based continuous reasoning to distribute container images in cloud-edge settings. ASP is also adopted by Le *et al.* (2017), where the authors addressed the distributed constraint optimisation problem. The constraint-based approach of Amadini *et al.* (2024) complements our solution by addressing sustainable cloud-edge application placement via adaptive service flavour and topology selection under cost and carbon constraints.

https://doi.org/10.1017/S1471068425100148 Published online by Cambridge University Press

Similarly to other declarative programming efforts, FlexiPlace allows modelling constraints including, for example hardware resources, availability, bandwidth, security policies, inter-component latency PUE, and carbon intensity. Differently from all previous work, it features the possibility of automatically relaxing lower-priority constraints when no feasible deployment can be determined. This flexibility goes beyond the state of the art, by implementing a graceful degradation of determined solution placements while guaranteeing critical constraints are met. To the best of our knowledge, FlexiPlace is the first approach integrating logic-based placement with priority-based constraint relaxation.

7 Concluding remarks

We proposed a declarative approach based on ASP for placing multi-service applications in cloud-edge environments, and its open-source prototype FlexiPlace. Our solution addresses satisfiable instances, allowing their declarative specification through a customisable and extensible taxonomy. Besides, it extends the state-of-the-art by solving unsatisfiable application placement instances through the selective relaxation of lower-priority constraints, according to priorities set by DevOps. Experimental results confirm the feasibility of the approach on realistic infrastructures (up to 500 nodes) and applications (up to 30 services). As future work, we plan to support more expressive requirements and integrate explanations for unsatisfiable placement instances by relying on the notion of *minimal unsatisfiable subprogram* (Alviano *et al.* (2023)).

Acknowledgments

This work has been partly supported by projects "*FREEDA*" (CUP: I53D23003550006), funded by the framework PRIN (Ministry of University and Research, Italy) and "SEcurity and RIghts In the CyberSpace - SERICS" (PE00000014 - CUP: H73C2200089001) under the National Recovery and Resilience Plan (NRRP) funded by the European Union - NextGenerationEU. DA is a member of the Gruppo Nazionale Calcolo Scientifico – Istituto Nazionale di Alta Matematica (GNCS-INdAM).

References

Aït-salaht, F., Desprez, F. and Lebre, A. 2021. An overview of service placement problem in fog and edge computing. *ACM Computing Surveys* 53, 3, 65:1–66:35.

Alviano, M., Dodaro, C., Fiorentino, S., Previti, A. and Ricca, F. 2023. ASP and subset minimality: Enumeration, cautious reasoning and MUSes. *Artificial Intelligence* 320, 103931.

Alviano, M., Dodaro, C., Leone, N. and Ricca, F. 2015. Advances in WASP. In *Logic Programming and Nonmonotonic Reasoning - 13th International Conference, LPNMR 2015, Lexington, KY, USA, September 27-30, 2015*, F. Calimeri, G. Ianni and M. Truszczynski, Eds. Lecture Notes in Computer Science, Vol. 9345, Springer, Lexington, KY, USA, 40–54. Proceedings

Alviano, M., Dodaro, C., Marques-Silva, J. and Ricca, F. 2020. Optimum stable model search: Algorithms and implementation. *Journal of Logic and Computation* 30, 4, 863–897.

Alviano, M. and Faber, W. 2018. Aggregates in answer set programming. *KI-Künstliche Intelligenz* 32, 2-3, 119–124.

Amadini, R., Gazza, S., Soldani, J., Vitali, M., Brogi, A., Forti, S., Giallorenzo, S., Plebani, P., Ponce, F. and Zavattaro, G. (2024) Pick a flavour: Towards sustainable deployment of cloud-edge applications. In *Logic-Based Program Synthesis and Transformation*, J. Bowles and H. Søndergaard, Eds. Springer Nature Switzerland, Cham, 117–127.

Andres, B., Kaufmann, B., Matheis, O. and Schaub, T. 2012. Unsatisfiability-based optimization in clasp. In *ICLP (Technical Communications)* LIPIcs, A. Dovier and V. Santos Costa, Eds., Vol. 17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 211–221. https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ICLP.2012.211.

Apat, H. K., Goswami, V., Sahoo, B., Barik, R. K. and Saikia, M. J. 2025. Fog service placement optimization: A survey of state-of-the-art strategies and techniques. *Computers* 14, 3, 99.

Azzolini, D., Forti, S. and Ielo, A. 2025. *Continuous Reasoning for Adaptive Container Image Distribution in the Cloud-Edge Continuum*. Cluster Computing. In press

Barabási, A.-L. and Albert, R. 1999. Emergence of scaling in random networks. *Science* 286, 5439, 509–512.

Baumeister, J., Herud, K., Ostrowski, M., Reutelshöfer, J., Rühling, N., Schaub, T. and Wanko, P. 2024. Towards industrial-scale product configuration. In *LPNMR*. Lecture Notes in Computer Science, Vol. 15245, Springer, 71–84.

Brewka, G., Eiter, T. and Truszczyński, M. 2011. Answer set programming at a glance. *Communications of the ACM* 54, 12, 92–103.

Brogi, A. and Forti, S. 2017. QoS-aware deployment of IoT applications through the fog. *IEEE Internet of Things Journal* 4, 5, 1185–1192.

Buccafurri, F., Leone, N. and Rullo, P. 2000. Enhancing disjunctive datalog by constraints. *IEEE Transactions on Knowledge and Data Engineering* 12, 5, 845–860.

Calimeri, F., Faber, W., Gebser, M., Ianni, G., Kaminski, R., Krennwallner, T., Leone, N., Maratea, M., Ricca, F. and Schaub, T. 2020. ASP-Core-2 input language format. *Theory and Practice Logic Programming* 20, 2, 294–309.

Calimeri, F., Fuscà, D., Perri, S., Zangari, J., Maratea, M., Adorni, G., Cagnoni, S. and Gori, M. 2017. I-DLV: The new intelligent grounder of DLV. *Intelligenza Artificiale* 11, 1, 5–20.

Erdem, E., Gelfond, M. and Leone, N. 2016. Applications of answer set programming. *AI Magazine* 37, 3, 53–68.

Erdős, P. and Renyi, A. 1959. On Random Graphs I. *Publicationes Mathematicae Debrecen* 6, 290–297. https://www.bibsonomy.org/bibtex/2420b83c1533188c0b54bd1f6eea2b782/krevelen.

Falkner, A. A., Friedrich, G., Schekotihin, K., Taupe, R. and Teppan, E. C. 2018. Industrial applications of answer set programming. *KI-Künstliche Intelligenz* 32, 2-3, 165–176.

Forti, S. (2022) Keynote: The fog is rising, in sustainable smart cities. In *PerCom Workshops 2022*. IEEE, 469–471.

Forti, S., Bisicchia, G. and Brogi, A. 2022. Declarative continuous reasoning in the cloud-IoT continuum. *Journal of Logic and Computation* 32, 2, 206–232.

Forti, S. and Brogi, A. 2022. Green application placement in the cloud-IoT continuum. In *PADL*, LNCS, Vol. 13165, Springer, 208–217.

Forti, S., Ferrari, G.-L. and Brogi, A. 2020. Secure cloud-edge deployments, with trust. *Future Generation Computer Systems* 102, 775–788.

https://doi.org/10.1017/S1471068425100148 Published online by Cambridge University Press

Gamblin, T., Culpo, M., Becker, G. and Shudler, S. 2022. Using answer set programming for HPC dependency solving. In *SC*, IEEE, 35:1–35:15.

Gebser, M., Kaminski, R., Kaufmann, B. and Schaub, T. 2011. Multi-criteria optimization in answer set programming. In *Technical Communications of the 27th International Conference on Logic Programming (ICLP'11)*, J. P. Gallagher and M. Gelfond, Eds., Vol. 11, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 1–10.

Gebser, M., Kaminski, R., Kaufmann, B. and Schaub, T. 2019. Multi-shot ASP solving with clingo. *Theory and Practice of Logic Programming* 19, 1, 27–82.

Gebser, M., Kaufmann, B., Neumann, A. and Schaub, T. 2007a. clasp : A conflict-driven answer set solver. In *Logic Programming and Nonmonotonic Reasoning, 9th International Conference, LPNMR 2007, Tempe, AZ, USA, May 15-17, 2007*, C. Baral, G. Brewka and J. S. Schlipf, Eds. Lecture Notes in Computer Science, Vol. 4483, Springer, 260–265. Proceedings

Gebser, M., Schaub, T. and Thiele, S. 2007b. Gringo: A new grounder for answer set programming. In *Logic Programming and Nonmonotonic Reasoning, 9th International Conference, LPNMR 2007, Tempe, AZ, USA, May 15-17, 2007*, C. Baral, G. Brewka and J. S. Schlipf, Eds. Lecture Notes in Computer ScienceLecture Notes in Computer Science, Vol. 4483, Springer-Verlag, Berlin, Heidelberg, 266–271.

Gelfond, M. and Lifschitz, V. 1991. Classical negation in logic programs and disjunctive databases. *New Generation Computing* 9, 3-4, 365–386.

Ghobaei-ARANI, M. and Shahidinejad, A. 2022. A cost-efficient IoT service placement approach using whale optimization algorithm in fog computing environment. *Expert Systems with Applications* 200, 117012.

Goudarzi, M., Palaniswami, M. and Buyya, R. 2023. A distributed deep reinforcement learning technique for application placement in edge and fog computing environments. *IEEE Transactions on Mobile Computing* 22, 5, 2491–2505.

Gupta, H., Vahid dastjerdi, A., Ghosh, S. K. and Buyya, R. 2017. iFogSim: A toolkit for modeling and simulation of resource management techniques in the internet of things, edge and fog computing environments. *Software: Practice and Experience* 47, 9, 1275–1296.

Le, T., Son, T. C., Pontelli, E. and Yeoh, W. 2017. Solving distributed constraint optimization problems using logic programming. *Theory and Practice of Logic Programming* 17, 4, 634–683.

Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S. and Scarcello, F. 2006. The DLV system for knowledge representation and reasoning. *ACM Transactions on Computational Logic* 7, 3, 499–562.

Mahmud, M. R., Srirama, S. N., Ramamohanarao, K. and Buyya, R. 2020a. Profit-aware application placement for integrated fog-cloud computing environments. *Journal of Parallel and Distributed Computing* 135, 177–190.

Mahmud, R., Ramamohanarao, K. and Buyya, R. 2020b. Application management in fog computing environments: A taxonomy, review and future directions. *ACM Computing Surveys* 53, 4, 1–43.

Massa, J., Forti, S. and Brogi, A. 2022. Data-aware service placement in the cloud-IoT continuum. In *SummerSOC – Revised Selected Papers. CCIS*, Vol. 1603, Springer, 139–158.

Massa, J., Forti, S., Paganelli, F., Dazzi, P. and Brogi, A. 2024. A declarative reasoning approach to conflict management in intent-based networking. In *ICIN 2024*, IEEE, 228–233.

Moreschini, S., Pecorelli, F., Li, X., Naz, S., Hästbacka, D. and Taibi, D. 2022. Cloud continuum: The definition. *IEEE Access* 10, 131876–131886.

Newman, M. E. J. 2010. *Networks: An Introduction*. Oxford University Press.

Niemelä, I., Simons, P. and Soininen, T. 1999. Stable model semantics of weight constraint rules. In *LPNMR*. Lecture Notes in Computer Science, Vol. 1730, Springer, 317–331.

https://doi.org/10.1017/S1471068425100148 Published online by Cambridge University Press

Pallewatta, S., Kostakos, V. and Buyya, R. 2023. Placement of microservices-based IoT applications in fog computing: A taxonomy and future directions. *ACM Computing Surveys* 55, 14s, 321:1–321:43.

Skarlat, O., Nardelli, M., Schulte, S., Borkowski, M. and Leitner, P. 2017. Optimized IoT service placement in the fog. *Service Oriented Computing and Applications* 11, 4, 427–443.

Smolka, S. and Mann, Z.á. 2022. Evaluation of fog application placement algorithms: A survey. *Computing* 104, 6, 1397–1423.

Soldani, J., Tamburri, D. A. and Van den Heuvel, W. 2018. The pains and gains of microservices: A systematic grey literature review. *Journal of Systems and Software* 146, 215–232.

Srirama, S. N. 2024. A decade of research in fog computing: Relevance, challenges, and future directions. *Software: Practice and Experience* 54, 1, 3–23.

Velepucha, V. and Flores, P. 2023. A survey on microservices architecture: Principles, patterns and migration challenges. *IEEE Access* 11, 88339–88358.

Vetriveeran, D. 2025. *Resource Provisioning in Fog Computing-a Survey*. ACM Computing Surveys.

https://doi.org/10.1017/S1471068425100148 Published online by Cambridge University Press