Declarative Deployment Planning for Green Pulverised Collective Computational Systems

Antonio Brogi[1], Roberto Casadei[2], Nicolas Farabegoli[2(✉)], Stefano Forti[1], and Mirko Viroli[2]

[1] University of Pisa, Pisa, Italy
{antonio.brogi,stefano.forti}@unipi.it
[2] University of Bologna, Cesena, Italy
{roby.casadei,nicolas.farabegoli,mirko.viroli}@unibo.it

Abstract. To promote non-functional goals (e.g., energy efficiency and reactivity) in system implementations, multiple strategies can be adopted, including the partitioning of distributed applications and the smart deployment of the resulting sub-components across the edge-cloud continuum. Within the aggregate computing approach to collective adaptive systems engineering (e.g., IoT ecosystems and robot swarms), the *pulverisation model* of partitioning and deployment works by splitting the collective computation into device computation rounds and in turn the device computation round in terms of five components: sensing, actuation, behaviour, state, and communication components. Previous research has investigated how different deployments of pulverised systems can provide different trade-offs involving performance and efficiency, with methodologies and simulation tools to carry out the comparison. However, there is still no contribution about the generation or search of effective deployments in the first place. To address this gap, this work introduces *Declarative Deployment Planning for Pulverised Systems (DePPS)*, an approach and toolchain based on simulation and a Prolog-based planner to guide the search of candidate deployments of pulverised systems. The benefits of the approach lie in its declarativity, modularity, scalability, and amenability for continuous reasoning of deployment alternatives. We exercise the approach with synthetic experiments and find out that we can achieve "greener" deployments (i.e., with low energy consumption and carbon footprint) while preserving good latencies compared to uniform peer-to-peer deployments.

Keywords: Collective systems · Edge-cloud continuum · Deployment · Pulverisation · Aggregate computing · Declarative programming

1 Introduction

General Background. The engineering of *collective adaptive systems (CASs)* [1,17,27] – cf. the Internet of Things (IoT) and swarm robotics – is

Published by springer Nature Switzerland AG 2025
C. Di Giusto and A. Ravara (Eds.): COORDINATION 2025, LNCS 15731, pp. 114–132, 2025.
https://doi.org/10.1007/978-3-031-95589-1_6

generally favoured by expressive programming models and techniques for efficient system implementation (e.g., optimised deployments and middlewares).

The *aggregate computing* approach [27] to CAS engineering, indeed, comprises a *macro-programming* model and languages [9] (based on *computational fields* [19,27]) for specifying self-organising behaviour and application partitioning and deployment models (such as the *pulverisation* model [6,7,13]) for flexible and efficient execution across the IoT-cloud continuum.

This programming model assumes a computational model where any device (i.e., any CAS participant) works at *sense–compute–interact* steps, which overall denote a single *round* of computation. Thus, an overall system execution can be modelled as a network of rounds (events) situated in space and time.

In the pulverisation model, an aggregate computing system is partitioned into small deployment units. The logical CAS (e.g., a swarm of robots) is conceptually split into individual devices (i.e., individual robots), and each individual robot is physically split into components that correspond to concerns of a single round of execution: i.e., sensing, actuation, state, computation, and communication components. In previous work [6,7,13], it was shown that different deployments (i.e., different mappings from components to physical hosts) can be simulated to get insights about different trade-offs in system execution (e.g., bandwidth consumption vs. latency vs. cloud costs).

Research Gap. Previous work has generally compared alternative deployments of pulverised systems without addressing the problem of "how to get a good deployment plan" in the first place. Essentially, much of the investigation was directed towards the assessment of "notable deployments" (e.g., fully peer-to-peer vs. uniform edge-/cloud-based) or manually defined variants thereof. Though the problem could be addressed by optimisation or constraint programming, e.g., by solving mixed-integer linear programming (MILP) problems or using satisfiability modulo theory (SMT) solvers, there are issues related to (i) *expressiveness*, as the complexity of deployment strategies may not easily be captured in purely mathematical/logical constraints, (ii) *implementation*, e.g., the handling of discrete and non-linear constraints (e.g., power consumption functions), (iii) *scalability*, as solving large problems/formulas is hard due to exponential complexity, and (iv) *operational aspects*, e.g., the difficulties related to supporting incremental reasoning and the handling of partial information.

Contribution. To tackle the above issues, in this paper, we propose *Declarative Deployment Planning for Pulverised Systems (DePPS)*, a methodology and toolchain that *integrates a Prolog-based deployment generator (planner) with the Alchemist pervasive computing simulator* [25] *to generate and assess deployments of pulverised systems*. Specifically, our contribution is threefold:

1. we present a declarative, Prolog-based planner for mapping multi-component pulverised systems to cloud-edge computing resources based on their non-functional requirements (including environmental sustainability);

2. we integrate the planner with the Alchemist simulator for pulverised systems, and use it for managing applications at runtime, enabling the system to react to dynamic changes in network conditions and resource availability;
3. our experimental evaluation shows that the we can generate "greener" deployment plans (i.e., with reduced energy consumption and carbon footprint) while still preserving acceptable latency performance compared to uniform peer-to-peer deployments (where each device hosts all the components).

We provide a companion software artefact [11], permanently archived on Zenodo, for reproducibility and inspectability of the toolchain and experimental setup.

Paper Structure. Section 2 provides background on pulverised aggregate computing systems. Section 3 describes the methodology and the Prolog-based planner. Section 4 evaluates the approach by presenting simulated experiments and results. Section 5 briefly reports on related work. Finally, Sect. 6 provides concluding remarks, and points out significant directions for further research on the topic.

2 Background: Aggregate Computing, Pulverisation and Prolog

Aggregate computing is the computational and programming model for CASs that motivates the deployment approach called pulverisation.

Aggregate Computing. Aggregate computing is a programming and computational model for collective behaviour. An aggregate computing *system* is a collection of *devices*. A device has *sensors* and *actuators* to interact with the environment, and is supposed to be able to exchange messages with a subset of other devices (*neighbours*). A device works in discrete asynchronous *rounds* of execution. Conceptually, every rounds consists of *sense–compute–interact* steps:

1. *sense*: the device gets a sample of its local context, by querying sensors and gathering valid (i.e. not expired) messages from neighbours;
2. *compute*: the device evaluates the "aggregate program" (which is the same for all the devices) against its local context, and the output of this evaluation consists of a final result value (which describes what actuations must be performed) and an *export* data structure to be shared with neighbours (supporting implicit coordination);
3. *interact*: the prescribed actuations are performed and the export data structure is sent to neighbours.

This paper does not cover how this programming model works, which is largely covered in more than ten years of research (cf. [9,27]). Rather, it focusses on the *deployment* of such systems in distributed infrastructures.

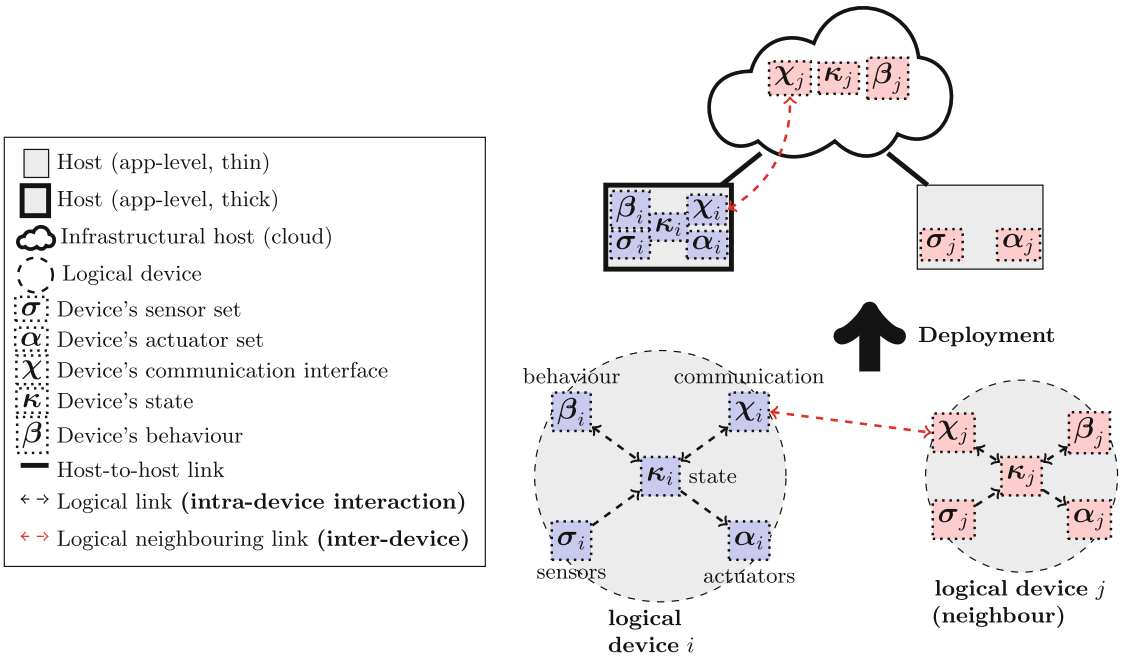

Fig. 1. A graphical idea of the pulverisation approach, considering two logical devices and three physical hosts. The upper part of the figure shows a hybrid deployment where one logical device is entirely deployed at one application-level device (thick host), and the other logical device gets partitioned, leaving only the sensing and actuation components at the thin host, and offloading the other components to the cloud. (Intra-device links between components are omitted in the deployment part of the figure.)

The *Pulverisation* Approach to Application Partitioning and Deployment. Term *pulverisation* [6,7,13] refers to an approach to partition aggregate computing systems into multiple *components* (deployment units).

Consider a swarm robotics application. We have a set of robots that we want to program or control as a collective. These robots are our *application-level devices*: each one of these should (conceptually) execute the aggregate computing rounds. In principle, we could deploy some aggregate computing *middleware* responsible for all the execution logic on each robot, and then distribute the aggregate program on them to run a swarm application. However, such a setup (which we call *fully peer-to-peer*) would not be flexible enough to accommodate heterogeneous systems and custom non-functional tradeoffs.

Indeed, we may have multiple purely *infrastructural devices* that are not part of the application per se but provide computational resources (e.g., edge, fog, and cloud servers). Furthermore, if our robot team is highly heterogeneous, we might have tiny robots (*thin devices*) that do not have enough computational power to compute, while other robots have enough resources to compute for other robots as well (*thick devices*). Thus, offloading parts of the computation of one (application-level) device to another physical device can enable tiny devices to participate in the system as well as flexible trade-offs of non-functional goals

(e.g., prioritising low carbon footprint and energy consumption rather than communication latency).

Pulverisation tackles this by splitting the device execution round into distinct concerns with corresponding deployment units: *sensing*, *actuation*, *state management*, *computation*, and *communication*. So, while the sensing and actuation components are generally deployed at the application-level devices (as they leverage the spatiotemporal situation to sample/affect different portions of the environment), the other components (state, communication, and computation) can be "disembodied" and hence may be deployed on other infrastructural devices (e.g., on the cloud). A notable deployment which we may call *(uniform) cloud-based* may consist of all the components deployed on the cloud, with the exception of sensing and actuation components at the application-level devices.

Previous research has analysed the trade-offs associated to different kinds of notable deployments (e.g., fully peer-to-peer vs. cloud-based vs. edge-based) [7,13], but *hybrid* deployment plans can also be generated, e.g., by taking into account the characteristics of individual devices, their communication technologies, design preferences, etc. Consider Fig. 1 for a graphical illustration of the pulverisation approach and the idea of a hybrid deployment (zooming on just a pair of neighbour devices—but keep in mind that the approach targets systems with dozens or hundreds of devices).

The Prolog Programming Language. As our solution relies on Prolog to determine eligible placements of pulverised systems, we here discuss some essential concepts about its syntax and functioning. Prolog is a declarative programming language based on first-order logic. Prolog programs consist of *clauses* (or *predicates*) of the form `a :- b1,...,bn.` stating that `a` holds if `b1` $\wedge \ldots \wedge$ `bn` holds. Clauses with empty premise (`n` $= 0$) are called *facts*. Predicate definitions can also contain *disjunctions* (denoted by ";") and negations (denoted by "`\+`"). Variables start with upper-case letters. Prolog programs can be queried, and the Prolog interpreter tries to answer each query by applying selective linear definite resolution and by returning a computed answer substitution instantiating the variables in the query. For instance, the query `?- nice(W).` on the program

```
nice(X) :- honest(X), gentle(X).

honest(alice).  honest(barbara).  gentle(barbara).
```

returns the computed answer substitution {`barbara / W`}, obtained by first rewriting the query by applying the first clause for `honest/1` and failing and then applying the second clause for `honest/1` and then the clause defining `gentle/1`.

3 Declarative Deployment Plans for Pulverised Systems

In this section, we describe our approach, **Declarative Deployment Planning for Pulverised Systems (DePPS)**.

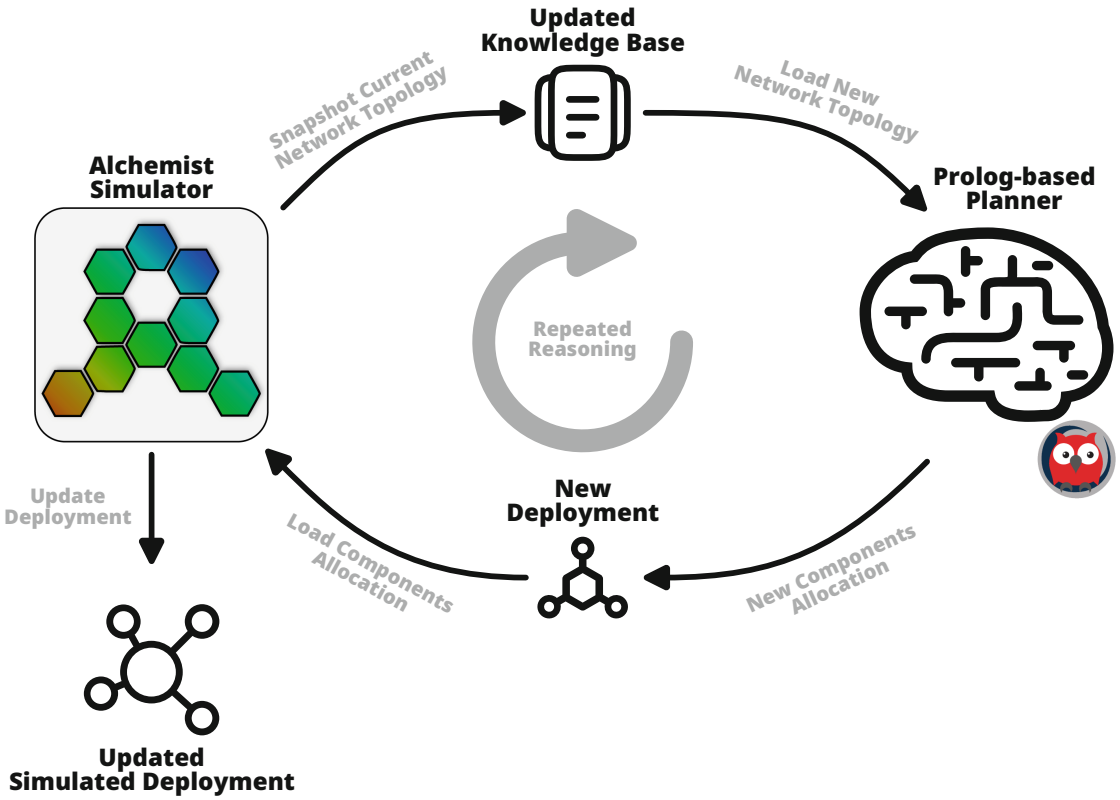

Fig. 2. Declarative Deployment Planning for Pulverised Systems (DePPS): architecture.

3.1 Architecture

The architecture of the approach is in Fig. 2. It is supported by a toolchain that includes the Alchemist simulator [25], a novel Prolog-based planner implemented in SWI Prolog [29], and tools supporting the Alchemist-SWI Prolog integration. The Alchemist simulator is a configurable simulator for pervasive computing systems, adopted extensively in aggregate computing research: it enables to set up an environment, a network of nodes, as well as the execution and communication logic for the system. Besides the integrated toolchain, the Prolog-based planner is the main novel contribution and is discussed in the next subsection (Sect. 3.2).

The methodology consists of iteratively invoking from the simulator the Prolog-based planner, endowed with heuristics for guiding the deployment plan generation, to produce up-to-date deployments—hence implementing a form of *"repeated reasoning"* on deployments to handle changes in network conditions. Further details on this methodology are provided in the evaluation discussed in Sect. 4.

3.2 Prolog-Based Planner

In this subsection, we cover the logic programming solution to solve the placement of pulverised applications to cloud-edge resources.

Knowledge Representation. A *digital device* is denoted by a unique identifier (UID) `DigDev`, the UID of its *knowledge* component `K` and a list of UIDs of its *sense* `S`, *act* `A`, *behaviour* `B` and *communication* `C` components. We model the above through facts like:

```
digitalDevice(DigDev, K, [S, A, B, C]).
```

The components are associated with hardware requirements `*HWReqs`, including a *maximum tolerated latency* `*MaxLatToK` towards the `knowledge` component[1], which their deployment must guarantee at runtime.

```
knowledge(K, HWReqs).
sense(S, SHWReqs, SMaxLatToK).
act(A, AHWReqs, SMaxLatToK).
behaviour(B, BHWReqs, BMaxLatToK).
communication(C, CHWReqs, CMaxLatToK).
```

On the other hand, physical devices that can support computation in the infrastructure are denoted by their UID `N`, available and total hardware capabilities – `FreeHW` and `TotHW` –, a list[2] of the `Sensors` and `Actuators` they can reach through local (wired or wireless) communication:

```
physicalDevice(N, FreeHW, TotHW, Sensors, Actuators).
```

Physical devices are associated with the energy source mix that powers them and a Power Usage Effectiveness (`PUE`) value, which represents the ratio between the total power absorbed by a computing system (including ancillary devices such as cooling) and the power it absorbs to only perform computation, through facts like:

```
energySourceMix(N, [(P1,EnergySource1), ..., (PK,EnergySourceK)]).
pue(N, PUE).
```

Energy sources that compose the mix of a node are associated with the portion of the mix they cover. A concrete example of an energy mix made of 30% solar energy and 70% coal energy is `[(0.3, solar), (0.7, coal)]`. Each energy source has an associated average *carbon intensity*, expressed in CO_2eq/kWh, and corresponds to the amount of greenhouse gas emissions produced per consumed unit of energy. The average carbon intensity of an energy mix is given by the

[1] Naturally, the knowledge component does not specify a maximum tolerated latency towards itself as such latency is always null.

[2] Sensors and actuators are each denoted through pairs like `(X, XType)` where `X` is the identifier of the sense or act component that will manage them at runtime and `XType` denotes the type of data they produce (e.g. temperature, brightness, video) according to some standard taxonomy such as [24].

```
1  place(DigDev, Nodes, Placement, I) :-
2      digitalDevice(DigDev, K, Components),
3      member((_,NK),Nodes), placeKnowledge(K,NK,KonN,I),
4      placeComponents(Nodes,Components,NK,[KonN],Placement,I).

5  placeKnowledge(K,NK,on(K,NK,HWReqs), I) :-
6      knowledge(K, HWReqs),
7      physicalDevice(NK, HWCaps, _, _, _),
8      ( member(used(NK,HWUsed), I); \+member(used(NK,_), I), HWUsed = 0 ),
9      HWReqs =< HWCaps - HWUsed.

10 placeComponents(Nodes,[C|Cs],NK,Placement,NewPlacement,I):-
11     member((_,N),Nodes), physicalDevice(N, HWCaps, _, Sensors, Actuators),
12     ( (sense(C, HWReqs, LatToK), member((C,_), Sensors));
13         (act(C, HWReqs, LatToK), member((C,_), Actuators)) ),
14     latencyOK(N,NK,LatToK),
15     hwOK(N,Placement,HWCaps,HWReqs,I),
16     placeComponents(Nodes,Cs,NK,[on(C,N,HWReqs)|Placement],NewPlacement,I).
17 placeComponents(Nodes,[C|Cs],NK,Placement,NewPlacement,I):-
18     member((_,N),Nodes), physicalDevice(N, HWCaps, _, _, _),
19     (behaviour(C, HWReqs, LatToK); communication(C, HWReqs, LatToK)),
20     latencyOK(N,NK,LatToK),
21     hwOK(N,Placement,HWCaps,HWReqs,I),
22     placeComponents(Nodes,Cs,NK,[on(C,N,HWReqs)|Placement], NewPlacement,I).
23 placeComponents(_,[],_,P,P,_).
```

Fig. 3. Declarative placement strategy.

average of the carbon intensities of the energy sources that make it out, weighted as per their portion of the mix they cover. Additionally, each node `N` is associated with a predicate `energyConsumption(N, Load, E)` that computes the energy consumption `E` of a node based on its current resource `Load`.

Last, but not least, end-to-end links between nodes `N1` and `N2` are denoted by their available `Latency` and `Bandwidth`, as in

```
link(N1, N2, Latency, Bandwidth).
```

Declarative Placement Strategies. Figure 3 lists the core predicates of the declarative strategy to determine eligible placements of pulverised applications, which we propose in this work. It is worth mentioning that it constitutes a declarative executable specification of *eligible placement* in our context. Prolog predicates are denoted as `pred/N`, where `pred` is their name and `N` their arity.

The `place/4` predicate (lines 1–4) returns a valid `Placement` of the digital device `DigDev` across a set of infrastructure `Nodes`, while also accounting for prior resource allocations within the considered infrastructure `I`. First, it selects a node `NK` $\in$ `Nodes` as the placement target for the knowledge component of the pulverised system (line 3). Then, the `placeKnowledge/4` predicate (lines 3, 5–9) gets the hardware requirements of this component (line 6) and checks the resources `HWCaps` at node `NK` (line 7) can support the new allocation of `K`, also accounting for resources already in use at `NK`, viz. `used(NK,HWUsed)` $\in$ `I` (lines 8–9).

After placing the knowledge component on node NK, the placeComponents/6 predicate (lines 4, 10–23) assigns the sense, act, behaviour, and communication Components, ensuring that they meet the required latency constraints towards the knowledge component. The first clause of placeComponents/6 (lines 10–16) is in charge of placing the sense and actuators components C onto nodes N ∈Nodes (line 11) checking that they can reach their required Sensors or Actuators, respectively (lines 12–13). Predicate latencyOk (line 14) checks that the end-to-end link supporting the communication between N and NK features a latency lower than the required LatToK. Subsequently, predicate hwOk/5 checks that cumulative hardware requirements of the current placement for node N leave enough free resources to support C, also factoring in previous resource allocation I (line 15). Last, placeComponents/6 recurs on remaining components Cs to be placed (line 16) extending the previous placement with the association between C and N, viz. on(C,N,HWReqs). The second clause of placeComponents/6 is analogous and places the behaviour and communication components checking latency (line 20) and hardware (line 21) requirements. A complete eligible placement is found when the list of components to be placed is empty, viz. [] (line 23). Recursion ends.

Environmental Footprint Assessment. Figure 4 lists the Prolog code used for estimating the environmental footprint of application placements. The footprint/4 predicate (lines 24–26) estimates the Energy consumption and Carbon emissions associated with a given Placement, as determined by place/4. To achieve this, it first retrieves all nodes involved in the specified Placement (line 25), removes duplicates through the sort/2 predicate, and then computes the overall estimate using the recursive predicate hardwareFootprint/5 (lines 26, 28–33). This latter predicate iterates through the list of spanned nodes, leveraging nodeEnergy/4 and nodeEmissions/3 to compute and accumulate the energy consumption and carbon emissions at each node, EnergyN and CarbonN, respectively.

The nodeEnergy/4 predicate (lines 30, 35–42) retrieves the information about the available FreeHW and total TotHW hardware capacity of each considered node N (line 36), along with any previous allocation to be considered in I. Based on that, it computes the node load OldL before allocating the currently considered Placement (line 38) and uses it to determine the associated energy consumption (line 39). It then sums up[3] the total amount of hardware to be allocated to Placement (line 40) and adds it to the figures to compute the new load NewL (line 41). Then, it computes the new energy consumption NewE. Finally, the energy consumption of Placement at node N is given by the difference between NewE and OldE, multiplied by the node PUE (line 42). Notably, some Placement instances may be virtually associated with null energy consumption, as they leverage surplus energy already consumed by previous allocations at the given node. This observation is particularly useful for achieving energy efficient resource allocation, as it highlights cases where new workloads can be deployed with minimal impact on overall energy consumption.

[3] The predicate sum_list(List,Sum) adds all the elements of List into Sum.

```
24  footprint(Placement,Energy,Carbon,I) :-
25      findall(N, member(on(_,N,_), Placement), Ns), sort(Ns, Nodes),
26      hardwareFootprint(Nodes,Placement,Energy,Carbon,I).
27
28  hardwareFootprint([N|Ns],Placement,Energy,Carbon,I) :-
29      hardwareFootprint(Ns,Placement,EnergyNs,CarbonNs,I),
30      nodeEnergy(N,Placement,EnergyN,I),
31      energySourceMix(N,Sources), nodeEmissions(Sources,EnergyN,CarbonN),
32      Energy is EnergyN + EnergyNs, Carbon is CarbonN + CarbonNs.
33  hardwareFootprint([],_,0,0,_).
34
35  nodeEnergy(N,Placement,Energy,I):-
36      physicalDevice(N, FreeHW, TotHW, _, _),
37      (member(used(N,UsedHW), I); \+member(used(N,_), I), UsedHW = 0),
38      OldL is 100 * (TotHW - FreeHW + UsedHW) / TotHW,
39      energyConsumption(N,OldL,OldE),
40      findall(H,member(on(_,N,H),Placement),HWs), sum_list(HWs,PHW),
41      NewL is 100 * (TotHW - FreeHW + PHW + UsedHW) / TotHW,
42      pue(N,PUE), energyConsumption(N,NewL,NewE), Energy is (NewE - OldE) * PUE.
43
44  nodeEmissions([(P,S)|Srcs],Energy,Carbon) :-
45      nodeEmissions(Srcs,Energy,CarbSrcs),
46      emissions(S,MU), CarbS is P * MU * Energy,
47      Carbon is CarbS + CarbSrcs.
48  nodeEmissions([],_,0).
```

Fig. 4. Declarative footprint estimate.

Similarly, given a node's energy mix and the energy consumption of a `Placement`, the `nodeEmissions/3` predicate (lines 31, 44–48) computes the carbon emissions. It calculates the weighted average of the carbon intensities `MU` of the available energy sources `S`, where each source contribution is weighed by its portion `P` in the overall energy mix (line 46). The results are accumulated in the `Carbon` variable (line 47), until the entire energy mix has been scanned (line 48).

Placing Multiple Digital Devices. The considered problem incurs in exp-time worst-case complexity as the input size grows, in terms of devices to be placed and nodes to be considered. This kind of problems has indeed been proved NP-hard [4]. Hence, we devised a heuristic strategy capable of placing multiple digital devices by exploring candidate placement nodes from those with lower carbon intensity to those with higher carbon intensity. Being a heuristic strategy, it drives the search towards potentially better candidate solutions with no optimality guarantees. While it practically reduces search times, it remains worst-case exp-time.

Particularly, it extends the behaviour of `place/4` (lines 1–4) by (*i*) preliminarily sorting candidate placement `Nodes` as per their average carbon intensity. Besides, it imposes maximum threshold values for the carbon intensity and energy consumption associated with the placement of a single node. Once a device has been placed, its allocation is added to the `I` variable by suitably updating `used/2` facts in `I` before attempting the placement of the next digital device. For the sake of brevity, we refer readers to the online open-source implementation of our approach for all details [11].

4 Evaluation

The proposed approach has been applied to different network topologies and a variable number of devices, demonstrating that, compared to notable deployments (e.g., the fully peer-to-peer deployment—cf. Sect. 2), we can (re-)generate deployment plans featuring improved carbon footprint and energy efficiency.

All experiments are conducted in Alchemist [25], a simulator for large-scale pervasive computing systems. The Prolog planner relies on the SWI Prolog [29] toolchain, specifically JPL[4], to integrate the Prolog solver with the simulator. The experiments are open-sourced under a permissive license on GitHub[5] and are permanently archived on Zenodo [11] for future reference and reproducibility.

Evaluation Goals. We aim to evaluate the proposed approach in terms of "green deployments", measured in terms of energy consumption and carbon footprint. Specifically, we aim to show that, compared with a fully peer-to-peer deployment, a sensible overall carbon footprint reduction of the system can be achieved while keeping, at the same time, acceptable system latencies.

4.1 Experimental Setup

We model a synthetic system that may be a proxy for a crowdsourcing application in a smart city or a swarm, with a dense set of devices moving around at moderate speed. Specifically, we consider a system composed of a variable number N of (thick) *physical devices* located in a two-dimensional space, with dimensions proportional to the number of devices. The devices are assumed to be thick, namely to be able to host the components to work autonomously; if they were thin, then offloading would be needed and not an option to improve non-functional concerns. The devices are randomly placed in a uniform distribution space. Each *physical device* is connected to other devices within a range of $2 \cdot \sqrt{N}$ meters, simulating a short-range communication medium like Bluetooth. This choice was made to maintain a proportional communication range across various scenarios with differing device counts, as the area on which the devices are distributed scales with the number of nodes. Additionally, a *cloud instance* is present in the environment and is connected to all *physical devices* in the system. Each simulated scenario runs for $t = 720$ steps, with each step representing a minute.

The connection link between devices in the simulation serves as a proxy for the communication latency of the medium. Each *physical device* moves freely in the 2D space according to a Brownian motion model. This movement introduces network perturbations in terms of both neighbourhood structure (i.e., available links and associated latencies), factors that the planner considers when searching for deployments.

[4] https://jpl7.org/.
[5] https://github.com/nicolasfara/experiments-2025-pulverization-prolog-placer.

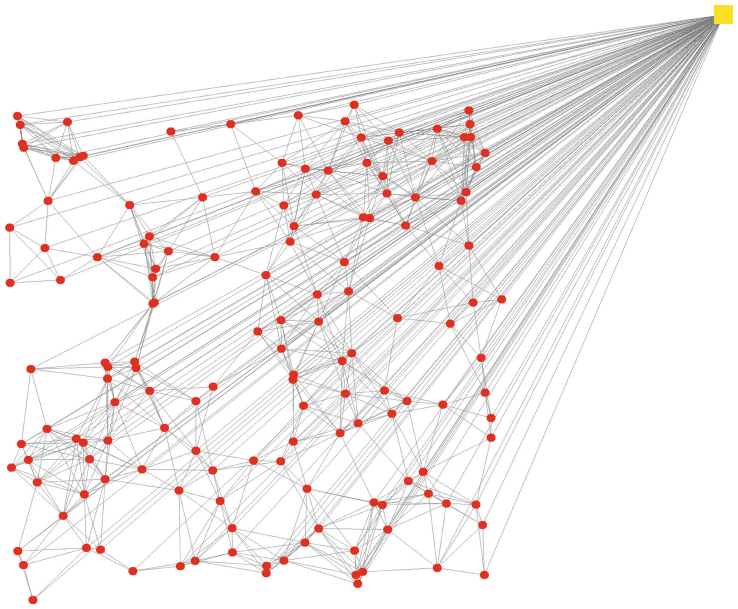

Fig. 5. Network of 150 *physical devices* (red circles) connected together within the range $2 \cdot \sqrt{N}$, and a *cloud instance* (yellow square) connected to all the devices. The gray lines represent the links between each device in the system. (Color figure online)

We evaluate two main scenarios: *device-only*, and *placer*. The first scenario acts as a baseline. The *device-only* scenario corresponds to the fully peer-to-peer deployment (cf. Section 2), where all the components for a logical device execute on a corresponding physical device. In the *placer* scenario, every 30 simulated minutes, the Prolog-based planner devises a new deployment, and the components are (re)deployed in the infrastructure accordingly. We suppose 30 simulated minutes to be a reasonable time interval for accounting network topologies variations in this system.

Each device has an energy source mix that follows a jittered sine wave pattern; in this way, we simulate a variable energy mix distribution to evaluate our planner in a dynamic condition. The sine wave driver is mainly adopted for its recurrent trend, and also because it is a good approximation for day-night energy mix. The maximum proportion of green energy is 90% for physical devices and 50% for the cloud instance. We repeated each experiment 10 times with different random seeds, and the results, along with their associated errors (see Sect. 4.2), are averaged over these repetitions to account for random fluctuations.

Figure 5 pictorially represents the setup described above.

4.2 Results

Energy and Carbon Footprint. The following analysis examines the *energy consumption* and *carbon footprint* of the system across the scenarios described in Sect. 4.1. Compared to the baseline scenario, the proposed approach achieves "greener" deployments (namely, lower energy consumption and lower levels of associated carbon emissions). Specifically, as illustrated in Fig. 6, given the same

network configuration, the deployment generated by the proposed approach consumes three times less energy than a traditional deployment (device-only).

Each physical device has a variable energy source mix driven by a sinusoidal pattern. This is evident in the *device-only* scenarios, where carbon emissions follow a sinusoidal trend. Since deployment remains unchanged in device-only scenarios, energy consumption remains stable throughout the simulation, and the carbon footprint reflects the aforementioned pattern. Intuitively, increasing the number of devices in the simulation results in a proportional increase in energy consumption and carbon footprint, as shown in the first row of Fig. 6.

In contrast, the proposed approach consumes less than 2 KWh with 50 nodes, increasing to 3.5 kWh with 100 nodes—more than three times lower than notable deployments. Additionally, since the planner is applied every 30 simulated minutes, it continuously recalibrates the deployment to adapt to network changes, such as new links between devices, and provides an a new eligible deployment, responding to infrastructural changes that might have affected the previous one. This is clearly reflected in the carbon emission trend, which does not precisely follow the sine pattern observed in the previous scenarios. Instead, the planner devises a new deployment that reduces the system's carbon footprint.

The proposed approach significantly reduces overall energy consumption and, consequently, the carbon footprint, ensuring a greener deployment regardless of network topology.

Inter-device and Intra-components Latencies. Latency is a sensible factor in large-scale self-organising networks. This analysis evaluates how the proposed approach affects system latencies compared to baseline deployments. Specifically, two latency categories are considered: *intra-component latency* and *inter-device latency*. *Intra-component* latency refers to the latency experienced by a single digital device when reaching its five components, which may be offloaded to other physical devices. *Inter-device* latency refers to the latency experienced by each communication component when communicating with the communication components of neighbouring digital devices.

The baseline scenario, in which the five components are executed on the same physical device, results in near-zero *intra-component* latency since component communication occurs in memory. In contrast, *inter-device* latency is proportional to the distance between interconnected physical devices. In a pulverised deployment generated by the planner, *intra-component* latency is expected to be higher than in the baseline scenarios. However, inter-device latency is expected to be similar to or even lower than that of the reference scenarios.

The experiments confirm this intuition, as illustrated in Fig. 7. The *intra-component* latency (represented by the blue line) is higher than in the device-only scenario. However, it is noteworthy that latency does not directly depend on the number of devices in the network. As the number of devices increases, intra-component latency remains within the same order of magnitude, indicating good scalability. This phenomenon is explained by the planner's deployment strategy,

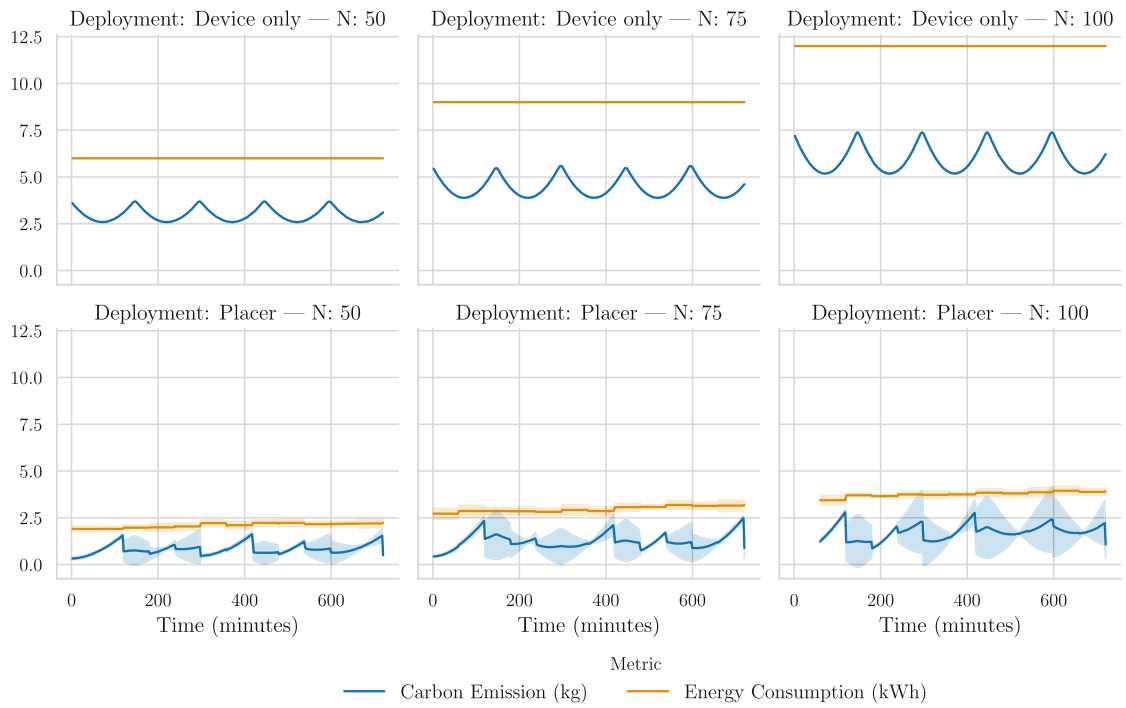

Fig. 6. Comparison between the "notable" deployment *device-only* (upper row) and the one produced by the Prolog planner (lower row). The orange line denotes the overall system energy (measured in kWh), while the blue line denotes the overall system carbon footprint (measure in kg). Notably, the deployments computed via Prolog planner result in a significant less carbon footprint, but also in a lower overall energy consumption, achieving greener deployment compared with the baseline. Shades show $\pm\sigma$ (stdev) which is almost null in the case of device-only placement. (Color figure online)

which tries to contain the number of devices used for component placement, thereby preserving locality regardless of the system's size.

Regarding *inter-device* latency, a slight but non-critical increase is observed. This increase occurs because the planner does not explicitly account for the communication component's location, treating it like other components. However, due to the locality-preserving nature of the planner, communication components are placed "nearby" other components, preventing a significant increase in latency between digital devices. Consequently, as the number of devices in the network increases, no substantial increase in latency is observed.

Though a slight latency increase is present, it has minimal impact on system performance. Crucially, the proposed approach greatly reduces energy consumption, resulting in greener deployments then fully peer-to-peer deployments.

For each simulated scenario, we extracted the time needed to compute the placement by the Prolog program. The execution time represents a valuable indicator of the effectiveness of the proposed approach, and its ability to scale with the number of devices. From the collected data, despite the number of simulated devices, the execution time is ~ 100 ms. The independence from the number of nodes, although we have simulated a limited variety of nodes, means

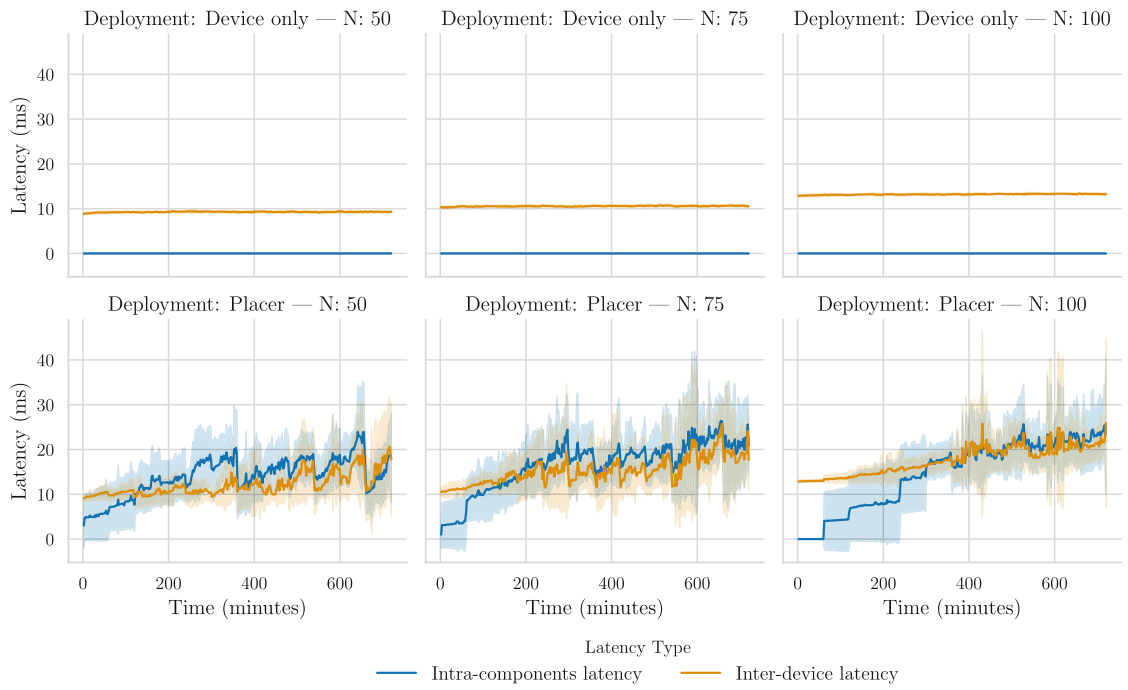

Fig. 7. Analysis between the two scenarios in terms of intra-components latencies, and inter-device latencies. The blue line represents the intra-components latency, namely the average latency measured to reach the other components instances offloaded in the other devices. The orange line represents the inter-device latency which measure the latency, for each communication component, with the other communication components in the neighbour devices. Shades show $\pm\sigma$. (Color figure online)

that the proposed approach promptly computes the solution, preserving margin for even bigger networks. The execution times are extracted using a consumer PC equipped with an AMD Ryzen 9 7900X (24) @ 5.733 GHz, and 64 GB of RAM.

5 Related Work

The focus of this paper is on the (re-)generation and assessment of deployment plans [2] for pulverised collective computation systems.

Broad Context: Component Models, Deployments, and Reconfiguration. The deployment of an application depends on how the application is partitioned into parts or components, which in turn depends on the adopted *component model* [10], with approaches ranging from *architectural description languages (ADLs)* [22] to *service-oriented approaches* [18]. Examples of component models include Fractal [5] and *BIP (Behaviour, Interaction, Priority)* [3]. A related topic is how to reconfigure existing deployments, e.g., for optimisation purposes or to withstand disruption. A good overview and survey on deployment reconfiguration is provided by Arcangeli et al. [2]. Examples of frameworks for

reconfigurable architectures include the rule-based framework *DReAM (Dynamic Reconfigurable Architecture Modeling)* [23] and DR-BIP [3]. These can be used, e.g., to formalise the pulverisation approach, which was described in [7] in terms of a structural operational semantics.

Pulverisation. The pulverisation idea and model were originally presented in [7], with a focus on the notion of "deployment independence": in principle, changing the deployment of a pulverised aggregate computing system does not affect the execution dynamics, and this can often be observed in practice. In [6], a methodology is described combining system descriptors, parametric deployment generation, and the EdgeCloudSim simulator [26] to assess static deployments. In contrast, the current paper investigates more deeply the deployment generation logic, sets up a reusable toolchain based on Prolog solvers (rather than simply parametric generators) and the Alchemist simulator (which is more suitable for CASs), and also performs analyses that take into account the energy and carbon footprint. In [13], a middleware and development platform for specifying and supporting the execution of pulverised systems is presented, together with the ability of enabling *dynamic reconfiguration* of the system following specified *local* rules. Such idea of using languages to express deployments is also investigated by so-called *multi-tier programming* languages [28], which use language features such as the type system to denote and check the placement of data and processes on different storage and computational *loci*. The use of *global* rules, expressed in the aggregate computing paradigm and leveraging patterns of collective adaptation [8], to induce "deployment self-organisation" is investigated in [12]. In this work, the dynamic reconfiguration is achieved through "repeated reasoning", namely by re-invoking the Prolog placer from time to time, hence exploring a different method to achieve effective deployments.

Declarative Approaches for the Generation of Deployment Plans. Prolog-based approaches have been proposed to address microservice application placement problems similar to the one examined in this article, focussing on different aspects, such as data awareness [20], security and trust constraints [16], environmental sustainability [15], and intent satisfaction [21]. Some recent proposals also employ continuous reasoning techniques [14] to streamline the generation of up-to-date deployment plans by integrating recent changes in the process without starting from scratch. Besides tackling a different problem than pulverised application deployment, these approaches are typically limited to solving the decision version of placement problems, without attempting to improve on target metrics as we do in this work.

6 Concluding Remarks

In this work, we propose Declarative Deployment Planning for Pulverised Systems (DePPS): an approach based on the use of a Prolog-based planner and

its integration with the Alchemist simulator to evaluate deployment plans for pulverised systems as well as the ability of reconfigurations to seek the desired trade-offs. To evaluate the approach, we run experiments in simulation, assessing that the planner can help to achieve "greener" deployments with low energy consumption and carbon footprint, hence contributing to the important thread of green and sustainable computing. As a by-product, we also provide a reusable toolchain (cf. the released artefact [11]) for reproducing the experiments and possibly assessing different kinds of deployments, generation strategies, etc.

Future work on this line is envisioned along multiple directions:

- **Continuous reasoning.** Currently, the reconfiguration is achieved by re-invoking the Prolog-based deployment planner at intervals with an up-to-date version of the deployment input knowledge. An interesting line is to extend the planner with "continuous reasoning" capabilities [14], enabling it to consider only what changed from the previous invocation and to incrementally produce up-to-date plans, also accounting for migration costs in terms of time.
- **Decentralised/hierarchical placement.** The current approach assumes a *centralised reasoner* invoked with up-to-date knowledge about the infrastructure. However, we could use the *divide-et-impera* principle and split the system into multiple management areas, and then use one reasoner for each to focus on a small part of the system. Previous work on the *Self-organising Coordination Regions (SCR) pattern* [8] could turn useful, as it enables to dynamically adjust the granularity of the system.
- **End-to-end simulation models.** Currently, the analysis conducted in Sect. 4 does not account for the overhead caused by the enaction of a deployment reconfiguration plan. Even though this does not invalidate the correctness of the deployment, aspects like convergence time to new deployments, and the communication overhead to migrate components are aspects that deserve attention in future work.
- **Generation of rules.** The current placer provides as output a deployment plan: an alternative is to generate local deployment rules, instructing individual devices about what deployment and re-configuration options they should follow and when (in what context).

References

1. Alrahman, Y.A., Nicola, R.D., Loreti, M.: Programming interactions in collective adaptive systems by relying on attribute-based communication. Sci. Comput. Program. **192**, 102428 (2020). https://doi.org/10.1016/J.SCICO.2020.102428
2. Arcangeli, J., Boujbel, R., Leriche, S.: Automatic deployment of distributed software systems: definitions and state of the art. J. Syst. Softw. **103**, 198–218 (2015). https://doi.org/10.1016/j.jss.2015.01.040
3. El Ballouli, R., Bensalem, S., Bozga, M., Sifakis, J.: Four exercises in programming dynamic reconfigurable systems: methodology and solution in DR-BIP. In: Margaria, T., Steffen, B. (eds.) ISoLA 2018. LNCS, vol. 11246, pp. 304–320. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-03424-5_20

4. Brogi, A., Forti, S.: QoS-aware deployment of IoT applications through the fog. IEEE Internet Things J. **4**(5), 1185–1192 (2017). https://doi.org/10.1109/JIOT.2017.2701408
5. Bruneton, E., Coupaye, T., Leclercq, M., Quéma, V., Stefani, J.: The FRACTAL component model and its support in java. Softw. Pract. Exp. **36**(11–12), 1257–1284 (2006). https://doi.org/10.1002/spe.767
6. Casadei, R., Fortino, G., Pianini, D., Placuzzi, A., Savaglio, C., Viroli, M.: A methodology and simulation-based toolchain for estimating deployment performance of smart collective services at the edge. IEEE Internet Things J. **9**(20), 20136–20148 (2022). https://doi.org/10.1109/JIOT.2022.3172470
7. Casadei, R., Pianini, D., Placuzzi, A., Viroli, M., Weyns, D.: Pulverization in cyber-physical systems: engineering the self-organizing logic separated from deployment. Future Internet **12**(11), 203 (2020). https://doi.org/10.3390/FI12110203
8. Casadei, R., Pianini, D., Viroli, M., Natali, A.: Self-organising coordination regions: a pattern for edge computing. In: Riis Nielson, H., Tuosto, E. (eds.) COORDINATION 2019. LNCS, vol. 11533, pp. 182–199. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-22397-7_11
9. Casadei, R., Viroli, M.: Declarative macro-programming of collective systems with aggregate computing: an experience report. In: Proceedings of the 26th International Symposium on Principles and Practice of Declarative Programming, PPDP 2024, pp. 5:1–5:5. ACM (2024). https://doi.org/10.1145/3678232.3678235
10. Crnkovic, I., Sentilles, S., Vulgarakis, A., Chaudron, M.: A classification framework for software component models. IEEE Trans. Softw. Eng. **37**(5), 593–615 (2011). https://doi.org/10.1109/TSE.2010.83
11. Farabegoli, N., Forti, S., Casadei, R.: nicolasfara/experiments-2025-pulverization-prolog- placer: 1.7.0 (2025). https://doi.org/10.5281/zenodo.14899534
12. Farabegoli, N., Pianini, D., Casadei, R., Viroli, M.: Dynamic IoT deployment reconfiguration: a global-level self-organisation approach. Internet Things **28**, 101412 (2024). https://doi.org/10.1016/j.iot.2024.101412
13. Farabegoli, N., Pianini, D., Casadei, R., Viroli, M.: Scalability through pulverisation: declarative deployment reconfiguration at runtime. Future Gener. Comput. Syst. **161**, 545–558 (2024). https://doi.org/10.1016/J.FUTURE.2024.07.042
14. Forti, S., Bisicchia, G., Brogi, A.: Declarative continuous reasoning in the cloud-IoT continuum. J. Log. Comput. **32**(2), 206–232 (2022). https://doi.org/10.1093/LOGCOM/EXAB083
15. Forti, S., Brogi, A.: Green application placement in the cloud-IoT continuum. In: Cheney, J., Perri, S. (eds.) PADL 2022. LNCS, vol. 13165, pp. 208–217. Springer, Cham (2022). https://doi.org/10.1007/978-3-030-94479-7_14
16. Forti, S., Ferrari, G., Brogi, A.: Secure cloud-edge deployments, with trust. Future Gener. Comput. Syst. **102**, 775–788 (2020). https://doi.org/10.1016/J.FUTURE.2019.08.020
17. Inverso, O., Trubiani, C., Tuosto, E.: Abstractions for collective adaptive systems. In: Margaria, T., Steffen, B. (eds.) ISoLA 2020. LNCS, vol. 12477, pp. 243–260. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-61470-6_15
18. Lemos, A.L., Daniel, F., Benatallah, B.: Web service composition: a survey of techniques and tools. ACM Comput. Surv. **48**(3), 33:1–33:41 (2016). https://doi.org/10.1145/2831270
19. Mamei, M., Zambonelli, F., Leonardi, L.: Co-fields: towards a unifying approach to the engineering of swarm intelligent systems. In: Petta, P., Tolksdorf, R., Zambonelli, F. (eds.) ESAW 2002. LNCS (LNAI), vol. 2577, pp. 68–81. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-39173-8_6

20. Massa, J., Forti, S., Brogi, A.: Data-aware service placement in the cloud-IoT continuum. In: Service-Oriented Computing - 16th Symposium and Summer School, SummerSOC 2022, Revised Selected Papers. Communications in Computer and Information Science, vol. 1603, pp. 139–158. Springer (2022). https://doi.org/10.1007/978-3-031-18304-1_8
21. Massa, J., Forti, S., Paganelli, F., Dazzi, P., Brogi, A.: Declarative provisioning of virtual network function chains in intent-based networks. In: 9th IEEE International Conference on Network Softwarization, NetSoft 2023, pp. 522–527. IEEE (2023). https://doi.org/10.1109/NETSOFT57336.2023.10175449
22. Medvidovic, N., Taylor, R.N.: A classification and comparison framework for software architecture description languages. IEEE Trans. Softw. Eng. **26**(1), 70–93 (2000). https://doi.org/10.1109/32.825767
23. De Nicola, R., Maggi, A., Sifakis, J.: DReAM: dynamic reconfigurable architecture modeling. In: Margaria, T., Steffen, B. (eds.) ISoLA 2018. LNCS, vol. 11246, pp. 13–31. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-03424-5_2
24. Pflanzner, T., Kertész, A.: A taxonomy and survey of IoT cloud applications. EAI Endorsed Trans. Internet Things **3**(12) (2017). https://doi.org/10.4108/eai.6-4-2018.154391
25. Pianini, D., Montagna, S., Viroli, M.: Chemical-oriented simulation of computational systems with ALCHEMIST. J. Simul. **7**(3), 202–215 (2013). https://doi.org/10.1057/JOS.2012.27
26. Sonmez, C., Ozgovde, A., Ersoy, C.: Edgecloudsim: an environment for performance evaluation of edge computing systems. Trans. Emerg. Telecommun. Technol. **29**(11) (2018). https://doi.org/10.1002/ETT.3493
27. Viroli, M., Beal, J., Damiani, F., Audrito, G., Casadei, R., Pianini, D.: From distributed coordination to field calculus and aggregate computing. J. Log. Algebraic Methods Program. **109** (2019). https://doi.org/10.1016/J.JLAMP.2019.100486
28. Weisenburger, P., Wirth, J., Salvaneschi, G.: A survey of multitier programming. ACM Comput. Surv. **53**(4), 81:1–81:35 (2020). https://doi.org/10.1145/3397495
29. Wielemaker, J., Schrijvers, T., Triska, M., Lager, T.: Swi-prolog. Theory Pract. Log. Program **12**(1–2), 67–96 (2012). https://doi.org/10.1017/S1471068411000494