Future Generation Computer Systems 161 (2024) 545–558

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Scalability through Pulverisation: Declarative deployment reconfiguration at runtime☆

Nicolas Farabegoli *, Danilo Pianini, Roberto Casadei, Mirko Viroli

Alma Mater Studiorum—Università di Bologna, Cesena, Italy

ARTICLE INFO

Dataset link: https://doi.org/10.5281/zenodo.11401482

Keywords:
Runtime reconfiguration
Distributed systems
Self-adaptation
Self-organisation
Pulverisation
Deployment

ABSTRACT

In recent years, the infrastructure supporting the execution of situated distributed computations evolved at a fast pace. Modern collective adaptive applications – as found in the Internet of Things, swarm robotics, and social computing – are designed to be executed on very diverse devices and to be deployed on infrastructures composed of devices ranging from cloud servers to wearable devices, constituting together a cloud–edge continuum. The availability of such an infrastructure opens to better resource utilisation and performance but, at the same time, introduces new challenges to software designers, as applications must be conceived to be able to adapt to changing deployment domains and conditions. In this paper, we introduce a practical framework for the development of systems based on the concept of *pulverisation*, meant to neatly separate business logic and deployment concerns, allowing applications to be defined independently of the infrastructure they will execute upon, thus supporting scalability. The framework is based on a domain-specific language capturing, in a declarative fashion: pulverised application components, device capabilities, resource allocation, and (runtime re-) configuration policies. The framework, implemented in Kotlin multiplatform and available as open source, is then evaluated in a small-scale real-world demo and in a city-scale simulated scenario, demonstrating the feasibility of the approach and its potential benefits in achieving better trade-offs between performance and resource utilisation.

1. Introduction

Recent technological and scientific advances are extending the *kinds of applications and systems* being addressed and their *supporting infrastructure* [1]. Specifically regarding infrastructure, it is mounting the idea of the *Edge–Cloud Continuum (ECC)* [2]: a multi-layer heterogeneous network of devices (ranging from large and powerful cloud servers to small connected *things*). There, software can compute, store, and exchange data in a distributed fashion while optimising for performance and resource utilisation.

This kind of infrastructure is particularly valuable for collective adaptive systems (CASs) [3–6], collections of devices and agents that interact to solve problems or provide services *cooperatively*, while adapting coherently "as a whole" to dynamic environments. Indeed, CAS implementations generally feature components that, depending on the conditions at hand, could benefit from being deployed on different devices or from *offloading* some of their tasks. Application examples of CASs include Internet of Things (IoT) deployments, swarms of robots, social computing systems, crowds of wearable-augmented people, and so on—supporting activities like monitoring, transportation, coordination, and other forms of collective intelligence [7].

However, exploiting this infrastructure poses new challenges to application designers. Commonly, applications are designed with a specific infrastructure in mind, whose assumptions unavoidably leak into the application logic [8] unless captured and encapsulated away from it: the higher the coupling between the application and the infrastructure, the harder it is for the former to exploit the latter's full potential and adapt to changes. The general solution is to devise a reasonable *partitioning* [9] of the software system and a corresponding (dynamic) deployment plan defining the mapping between the software components and the target deployment domain [10]. The central idea is to *decouple* the application logic from the deployment concerns, allowing the former to be designed as independently of the latter as possible.

In this work, we investigate whether existing techniques born in the context of CASs can be adapted to the cloud–edge continuum, and

☆ This work has been supported by the Italian PRIN Project COMMON-WEARS (2020HCWWLP).

* Corresponding author.
E-mail addresses: nicolas.farabegoli@unibo.it (N. Farabegoli), danilo.pianini@unibo.it (D. Pianini), roby.casadei@unibo.it (R. Casadei), mirko.viroli@unibo.it (M. Viroli).

https://doi.org/10.1016/j.future.2024.07.042
Received 30 November 2023; Received in revised form 7 June 2024; Accepted 22 July 2024
Available online 25 July 2024
0167-739X/© 2024 The Author(s). Published by Elsevier B.V. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

how they can be extended to support scalability and achievement of specific performance trade-offs. In particular, previous work proposes an application partitioning schema, called *pulverisation* [6,11], that fosters the decoupling of business logic and deployment concerns. In short, the core idea is to consider devices as *logical* entities whose software can be designed as (or broken down to, if already existing) an ensemble of *five components (behaviour, state, communication, sensors, and actuators)* that can be deployed with flexibility on available infrastructure without (ideally) affecting application functionality. We call a *pulverised system* a system that is partitioned according to this schema. This way, the designer can focus on the business logic at hand, delaying the definition of an optimised deployment to later stages of the software design, thus gaining resilience to changes in the infrastructure or in non-functional requirements. Although interesting, pulverisation has been so far limited under the point of view of dynamicity, ultimately hindering its capability to scale with mutating conditions or changing requirements: although the application can be deployed on arbitrary systems, there is no support for the reconfiguration of components at runtime.

In this work, we improve over the state of the art by introducing dynamicity in pulverised systems, retaining the neat separation between functional and non-functional specifications while providing means to tackle scalability. Specifically, we provide three main contributions:

1. we extend the theoretical framework of pulverisation by adding support for *runtime configuration rules*, allowing the system to *adapt at runtime* by moving pulverised components across the available infrastructure;
2. we accordingly provide a practical implementation of reconfigurable pulverisation called PulvReAKt, developed in the Kotlin programming language; and
3. we validate the proposed framework and approach both by simulation and through a real deployment, and provide open-source archived artefacts [12,13], useful for reproducibility and as base for future work.

In particular, with respect to previous work [11], which is mainly theoretical, we provide as major novel contributions: (i) two DSLs for *specifying* pulverised systems and deployment reconfiguration rules, (ii) a working implementation of a *runtime* (or *middleware*) supporting pulverisation *and* reconfiguration, and (iii) experiments on *runtime adaptation* of pulverised systems (where in [6,11] these are only statically generated).

The manuscript is organised as follows. Section 2 provides an explicit account of the research questions. Section 3 provides background on deployment and pulverisation. Section 5 describes the proposed DSL and platform. Section 7 provides an evaluation of the approach. Section 4 covers related work. Finally, Section 8 concludes the paper and highlights directions for future work.

2. Motivation and research questions

This work draws motivation from the problem of *deployment reconfiguration* for CASs, as a way to dynamically improve the non-functional profile of these kinds of systems by leveraging additional infrastructure as provided by the edge–cloud continuum. Previous work [11] has shown that *pulverising* the responsibility of individual devices into common components (covering sensing, actuation, state, computation, communication) provides flexibility at the deployment stage, and that deployments can be generated statically and then evaluated by simulation [6]. However, when the infrastructure is subject to highly dynamic conditions (due e.g., to changing load, energy availability, or network status) as in the case of the ECC, the deployment plan may need to be adapted to *scale* the application and improve performance and resource utilisation. Thus, the focus of this work is on *the specification and runtime support of* ***reconfigurable*** *pulverised systems*; in particular, the focus is *not* on illustrating the benefits of the pulverisation approach with respect to other component models (such as those reviewed in Section 4.1).

Specifically, with this work, we mean to answer the following research questions:

(RQ1) **How can a pulverised deployment be reconfigured at runtime in a decentralised way to adapt to changing conditions (load, energy, etc.)?**
(RQ2) **How can such adaptation be designed to provide relevant benefits over a static allocation?**

To answer these questions, this work advances the current state of the art on pulverisation approaches by introducing runtime reconfiguration rules and providing a practical implementation of a reconfigurable pulverisation framework.

3. Background

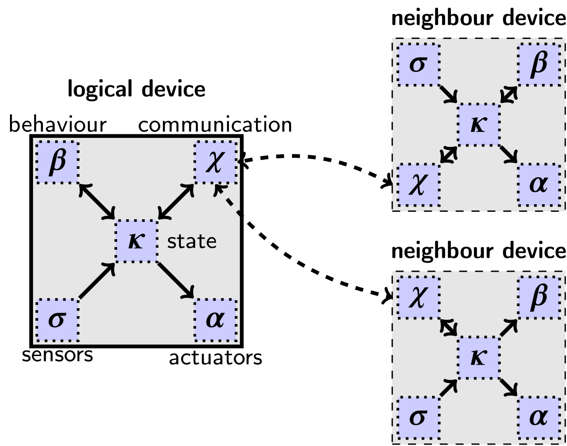
In this background section, we provide a conceptual framework for deployment and reconfiguration (Section 3.1) and then describe the *pulverisation* approach to application partitioning and deployment (Section 3.2).

3.1. Deployment and reconfiguration: Basic concepts

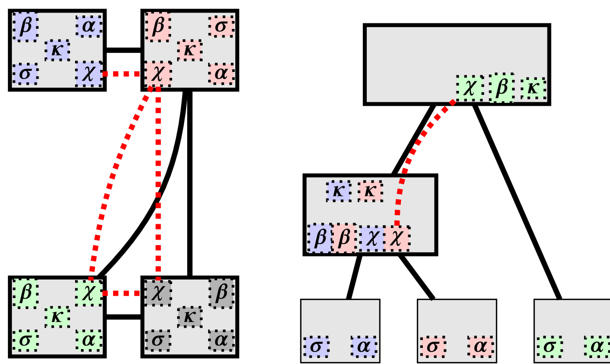
The *deployment view* is a well-known architectural viewpoint for software systems, concerned with the mapping of *software components* to *physical machines* [14] and supported by modelling notations like *UML Deployment diagrams*. Here, we briefly introduce deployment and reconfiguration based on the conceptual characterisation of [10,15]. A *site* is a set of computers (*hosts*) that may host a *software system*, i.e., a product described by a coherent collection of artefacts and consisting of a set of *deployment units* that can be independently operated. *(Software) Deployment* is the process of moving and making a software system available and operational from one or more *producer sites* to a target set of *consumer sites*, also called the *deployment domain*. The *deployment plan* defines the mapping between the software system and the deployment domain, possibly augmented with further information (e.g., metadata, constraints, preferences). Common deployment-related activities include: (i) *release/update* of the software system at the producer site; (ii) *installation/uninstallation* at/from the consumer sites; (iii) *activation/deactivation*, for starting/stopping the components; (iv) *reorganisation* of the software system; and (v) *redistribution*, i.e., changing the deployment plan. According to the analytical framework of [10], the problem of (automatic) deployment of distributed software systems can be addressed by considering (i) the nature of the software to be deployed (e.g., how the software system is split into components); (ii) the nature of the deployment domain (i.e. the characteristics and topology of the available infrastructure); (iii) how the deployment is designed (e.g., how the deployment plan is specified); and (iv) how the deployment is performed (e.g., how deployment activities are carried out). From an operational point of view, deployment may be supported by a so-called *runtime* or *middleware* [16]. Related work covering these issues is provided in Section 4. In the following, we introduce the deployment approach that we extend and upon which we develop PulvReAKt.

3.2. Pulverisation

Pulverisation [6,11] is an approach to distributed application partitioning and deployment, exemplified in Fig. 1 and described in the following. Its goal is to provide application designers with a way to specify the functional semantics of their software in a deployment-independent way. To do so, the application logic should be designed considering a *logical system*: a network of *logical* devices forming a network with *arbitrary* topology. A software system is pulverisable if the application of every single logical device is decomposable into an ensemble of *pulverised components* representing, respectively:



(a) A logical device, split into sub-components, and two of its neighbours.



(b) Peer-to-peer architecture: one-to-one mapping between logical and physical devices, with no offloading.

(c) IoT hosts can be thin, with some components offloaded at the edge and cloud.

Fig. 1. Pulverisation model and examples of deployments. Notation: solid-border boxes denote physical hosts (bold borders are for thick devices); solid lines denote connections between hosts; dashed-border boxes denote software components; different colours denote (software components of) different logical devices; red dashed lines denote connections between the software components, i.e., neighbouring relationships (not shown for co-located software components).

- a set σ of logical sensors;
- a set α of logical actuators;
- a state κ, representing the logical device's knowledge;
- a communication component χ, handling interaction with reachable devices in the logical system, and
- a computation component β, modelling the behaviour of the logic device.

Decomposition of an application into pulverised components can be achieved in two ways: either the application is designed with pulverisation in mind, or the application is developed using a framework supporting automatic decomposition (one notable example are *aggregate computing* frameworks [17,18]). Once the application has been pulverised, a mapping must be provided between the pulverised logical system and the hosts that will execute the pulverised components. In this process, a single logical device could (and usually does) end up being executed on multiple hosts. Indeed, though logical devices are generally associated with "application-level" physical devices that need to be controlled or monitored (e.g., a drone, a sensor, or a person—a logical ensemble of wearables), their execution can also be supported by other physical devices (e.g., purely infrastructural ones).

For instance, let us assume a minimal logical system counting two devices: two rain gauges logging the water level for future reference and opening a valve when the water level crosses a threshold in both devices. Once pulverised, the logical system would be split into ten deployable pulverised components (that we indicate with their component symbol and an index, e.g., σ_1 is the logical sensor of the first device). These pulverised components can be deployed in arbitrary hosts as far as they have the *capabilities* required to host them (for instance, the device executing the rain gauge logical sensor must be equipped with the right sensor, while the host executing the behaviour must be sufficiently powerful to execute the logic). Thus, with *no change to the application logic*, the system could be deployed on very different systems: let us assume, for instance, that our actual target infrastructure is not composed of two connected devices but is an existing IoT system where the rain gauges are connected to two LoRaWAN motes (which are too weak and energy-critical to execute the logic), a dedicated device controls the valves, the levels must be logged on a database hosted on a cloud server reachable via HTTPS, and we have an internal edge server that we can use as we please. In this case, we would deploy σ_1 and σ_2 on the motes, α_1 and α_2 on the valve controllers, κ_1 and κ_2 on the cloud server, and the remaining components on the edge server.

Although the idea behind pulverisation is simple, finding a way to implement it effectively requires tackling several challenges at different levels:

- **communication**: splitting logical devices into small deployment units implies communication among them, thus requiring to consider networking at two levels: across pulverised components (intra-device) and among logical devices (inter-device);
- **portability**: hosts will have very diverse hardware specifications, operating systems, and software stacks;
- **runtime**: the system should be able to reconfigure its deployment at runtime without disrupting the application logic; and
- **language**: for the idea to be exploitable by designers, it is essential that the pulverised configuration and the mapping of pulverised components to the underlying infrastructure can be expressed easily and, possibly, declaratively.

4. Related work

Following the conceptual framework introduced in Section 3.1, in this section, we cover related work about languages for specifying distributed systems (Section 4.1) and languages for specifying infrastructures, deployments, and reconfigurations (Section 4.2).

4.1. Application description languages: Component-based software engineering and the pulverisation model

We consider a distributed application as a graph of deployable units. This partitioning may be manually specified at development time (e.g., by explicitly defining and packaging different components) or automatically defined through *application partitioning* approaches [9].

In our approach, the application partitioning into components is manually specified through a DSL. Therefore, it can be framed in the context of *component-based software engineering*, with [19] providing a comprehensive review. The specification or design of distributed systems may leverage component models [20], architectural description languages [21], service-based compositions [22], or frameworks.

Specifically, this work builds on the *pulverisation* component model [6,11], whereby a large-scale cyber–physical system is partitioned into a graph of devices where each device is split into five deployable components: (i) sensors interface, (ii) actuators interface, (iii) behaviour, (iv) state, and (v) communication component. In [6], a methodology on top of the pulverisation model is proposed for generating and testing deployments through simulation. There, file descriptors and scripts are

used to generate deployment plans; in PulvReAKt, instead, we provide DSLs supporting a specification-oriented, declarative approach.

Additionally, the approach of specifying in a single codebase parts of the structure, behaviour, interaction and/or other aspects of a whole distributed system is also related to *macroprogramming* [23] and, in particular, to *multi-tier programming* paradigm [24], where the deployment units for different tiers (e.g., client and server tiers; or view, business logic, and data tiers) are obtained by compilation or interpretation of a single codebase. PulvReAKt adopts the same idea, where a logically centralised specification of the distributed architecture enables runtime checks and deployment activities.

4.2. Infrastructure and deployment description languages

A *deployment plan* is a configured mapping of a software system onto a deployment domain. Given a deployable software system developed using the techniques of the previous subsection, what is needed now is a language to describe a deployment domain and the deployment mapping. Languages have been proposed targeting specific deployment domains such as the smart grid (cf. *dspec* in the *RIAPS* platform [25]) or the cloud (cf. the *CAMEL* multi-DSL [26]). In this work, we are especially interested in deployments over large-scale cyber–physical systems and the edge–cloud continuum. The main aspects distinguishing PulvReAKt from other approaches are that it targets *pulverised systems* (and not general partitionings) and that it uses internal DSLs embedded in Kotlin.

A related (external) DSL is *MuScADeL* [27]: it allows expressing deployment properties of applications on multi-scale deployment domains, i.e., large and heterogeneous infrastructures considered under several viewpoints (device, geography, network, administrative domains, etc.). Component descriptions can include *constraints* (both basic software/hardware constraints and multi-scale criteria). The specifications are used to generate *probe* artefacts supporting data collection at the level of the deployment middleware. However, the work in [27] is preliminary and does not cover runtime adaptation.

A comprehensive conceptual framework on automatic deployment of distributed systems is provided in [10]. A more recent survey [28] focuses on formal techniques for verifying the correctness of reconfigurations in component-based distributed software systems. Indeed, formal models can help specify reconfigurable architectures: examples include *DR-BIP* [29] and *DReAM* [30]. Both are based on the same conceptual model: they feature *components* (capturing behaviour), *connectors* (capturing the interaction between components' *ports*), *maps* (logical topologies), and *deployments* (associating components to map locations), overall organised in *motifs* (dynamic architectural configurations), to model and analyse dynamic architectures. These tools can support *reasoning* about reconfigurations and *verification*.

Several approaches use languages together with middlewares to specify and execute reconfiguration policies. The *AWaRE DSL* [31] supports constraint-based self-management, leveraging managing agents. The language enables the specification of the domain model (in terms of components), the problem structure model (in terms of constraints), the agent architecture model (in terms of agents, roles, and their coordination), and the assignment model (in terms of management problem assignment strategies). Similarly, earlier approaches based on the specification of *constraint-sets* include *DELADAS* [32] and *ConfSolve* [33]. *Ctrl-F* [34] is an architectural description language with constructs for specifying adaptive behaviour and policies (constraints) for reconfiguring system components. The idea of its DSL to reconfiguration is to let designers specify behaviours that regulate the transitions between explicitly specified *configurations*. In [35], a model-based deployment approach is proposed targeting so-called *fleets*, i.e., distributed and heterogeneous devices at the edge characterised by different cyber–physical contexts (cf. resources, connectivity, etc.). The main goal is to support the distribution of *multiple variants* of the same application based on the different contexts of the different target devices.

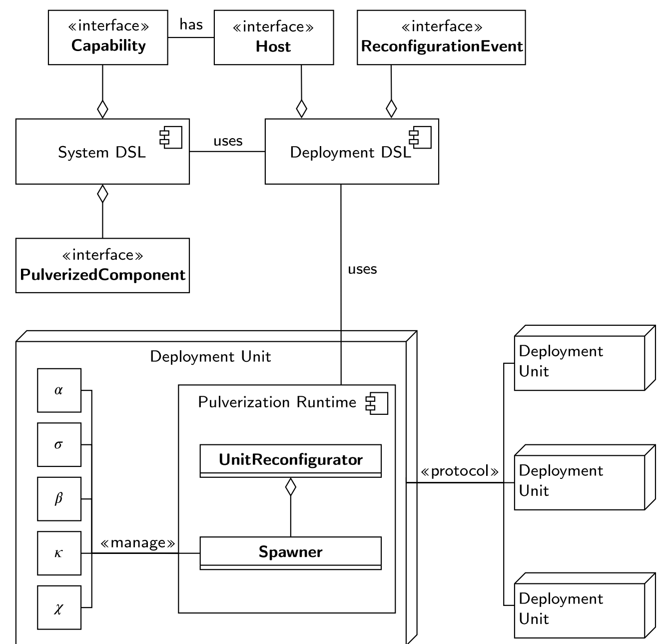


Fig. 2. The approach provides two DSLs and a runtime for specifying and executing deployment plans. Each deployment unit will be delivered to the target site and properly configured to interact with other deployment units according to the protocol that the application at hand requires (see Section 7 for examples).

The approach is based on the *GeneSIS* modelling language, where a `DeploymentModel` consists of sets of `Resources` (e.g., `Components`, specialised by `InfrastructureComponents` and `SoftwareComponents`) with associated `Propertys` and `Links`. However, these approaches, being based on constraint solving, typically suffer from scalability issues.

Finally, there is a plethora of approaches for automatic deployment or offloading across the edge–cloud continuum. One example is *osmotic computing* [36], whereby microservices (the solvent) can migrate across the edge–cloud infrastructure (the solution), passing through layer boundaries (the semi-permeable membranes) in order to keep a balance in the desired properties (the solute).

5. A runtime and DSLs for reconfigurable pulverised systems

The original pulverisation approach, as described in [11], does not provide any support for the dynamic relocation/reconfiguration of the pulverised components, a significant limitation when scalability is required. Thus, in this section, we present two main contributions to the extension of the pulverisation approach to support scalability: first, we clarify how we support the dynamic reconfiguration of pulverised systems (Sections 5.1, 5.2); then, we cover the two DSLs for specifying pulverised systems and their reconfiguration logic (Sections 5.3 and 5.4); and finally, we provide details about the middleware architecture and logic (Section 5.5).

An overall view of the approach in terms of the provided tools and the main modelling concepts is provided in Fig. 2. The elements depicted in the figure are explained in the following sections.

5.1. Pulverisation with dynamic reconfiguration

To support the dynamic relocation of the execution of pulverised components, we extend the pulverisation approach introducing the concept of *capability*. Capabilities manage the way pulverised components are deployed on the hosts and how (if) they should relocated at runtime.

In the proposed approach, each host exhibits a set of capabilities, representing features available to the pulverised components, such

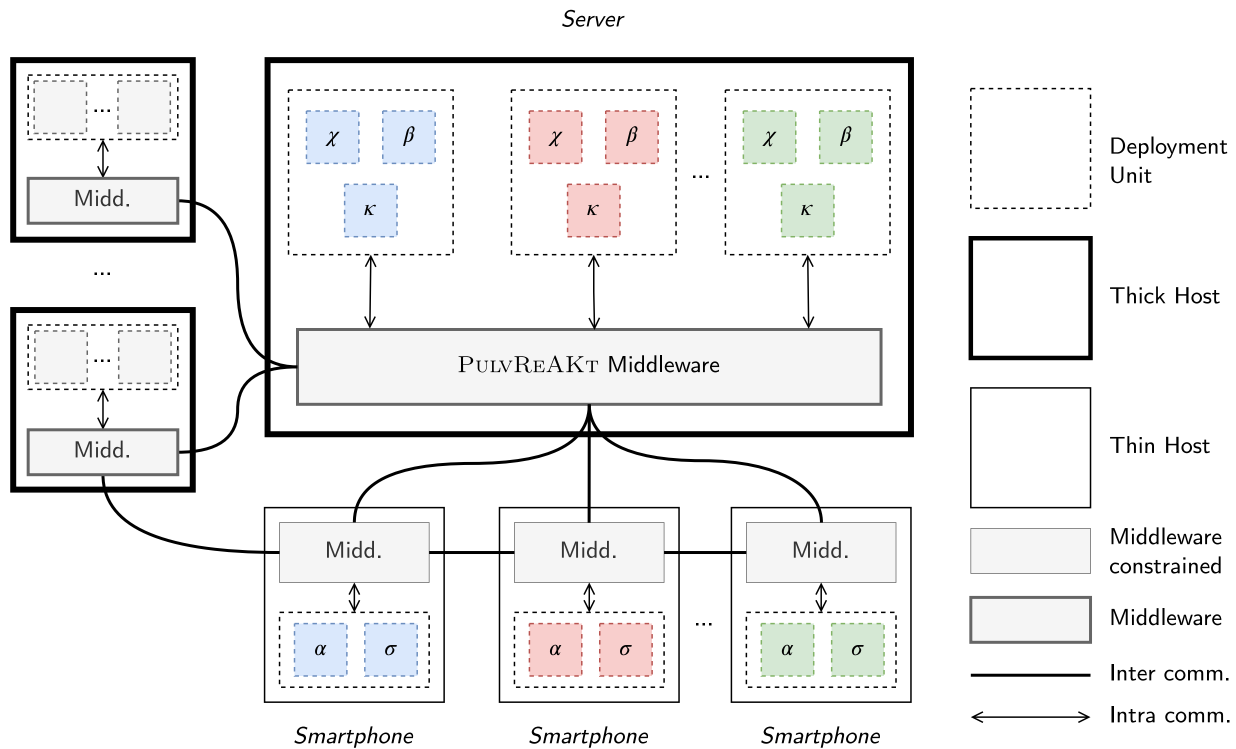


Fig. 3. A general example of a PulvReAKt system deployment. The figure shows how the system is *decentralised*, running a middleware instance on each node. Notice that we abstract the details about how the middleware instances are connected, which in general depends on the physical network connection, which is managed by a dedicated middleware component called `Communicator`.

as hardware resources, specific configuration settings, and userland policies. Complementarily, each pulverised component requires a set of capabilities, intended, from this perspective, as the features that the host must provide to the pulverised component for it to be executed. Following this approach, akin to labelling systems found in container orchestration platforms, a pulverised component can be deployed on a host that provides a superset of the capabilities required by the pulverised component.

Introducing the concept of *capability* in the original pulverisation approach enables the definition of fine-grained requirements for pulverised components, thus preventing illegal deployment combinations, even when these are the result of changing runtime conditions. For instance, the deployment of an α component on a host that does not provide any sensing capability will not be allowed; and, interestingly, a β component deployed on a host whose performance degraded too much (e.g., due to aggressive CPU throttling in response to a critical battery level) may be relocated elsewhere, making the improved approach more robust to changes introduced after the validation of the initial deployment plan.

In the remainder of the paper, we will refer to devices capable of hosting only *sensors* (σ component) and *actuators* (α component) as *thin*, since they are supposed to be resource-constrained devices not capable of executing the remainder of the pulverised components. On the other hand, we will refer to devices capable of hosting the β, χ, κ components (beside possibly the σ and α components) as *thick*, since they are supposed to be powerful enough to execute the logic of the pulverised components. Crucially, a pulverisation middleware must be capable of running on both devices, possibly using a stripped-down, lightweight version for thin devices.

To support constrained devices that can only host sensors and actuators, we consider a lightweight variant of the middleware (refer to Fig. 4). Compared to the full-fledged version running on thick devices, the lightweight variant does not implement the `UnitReconfigurator`, since there is no need to relocate components on thin devices, as sensors and actuators are supposed to be fixed in the (thin) host. The other main difference is that, since the β, χ, and κ components are not supported on thin devices, there is no need for embedding software libraries (such as the Aggregate Computing runtime) for behaviour implementations. Such a lightweight middleware variant can be also implemented using languages more suitable for resource-constrained devices, such as C/C++ or Rust, enabling a wide range of target devices otherwise impossible to support with a full-fledged middleware implementation.

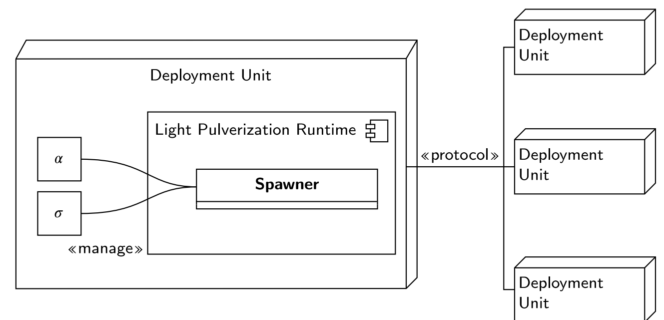


Fig. 4. Lightweight middleware implementation for *thin* devices. This version of the middleware does not support the reconfigurations and can execute only the σ and α components.

When lightweight middleware instances are involved in the system, at least one thick device must be available in the infrastructure to be able to execute the remainder of the components required by the application logic. Via the capabilities mechanism, we can ensure that invalid deployment configurations are prevented, and the system can be reconfigured at runtime to adapt to changing conditions.

Additionally, the capability mechanism can be used to enforce the typical "pairing" of the σ and α on the same physical device, preserving the situatedness of the application logic. Conversely, PulvReAKt can

support multiple sensors (or actuators) to be deployed on different physical devices, but still belonging to the same logical device, to enable a variety of heterogeneous deployment scenarios. In Fig. 3 is reported an example of a system running the PulvReAKt middleware composed of three thin and a single thick host. The figure shows how the middleware is deployed on each node of the infrastructure and how the *deployment units* are managed inside each host. In the figure, each middleware of the *thin* host is directly connected to the *thick* host. Although the image does not depict the physical connection between the hosts but only the logical ones among the middleware instances, the framework is not limited to a specific topology, as a dedicated middleware component (the `Communicator`) manages the construction of a routing table over physical network connection.

5.2. On middlewares and (Domain-specific) languages for deployed systems specification and execution

As mentioned in Section 3.1, the deployment and reconfiguration of a software system may be supported – following the well-known layered architectural style – by a *runtime* or *middleware* [16] encapsulating, among others, deployment and reconfiguration services. The runtime could be reified into a deployment unit *per se* or can be implemented as part of the deployment units of the software system to be deployed—e.g., taking the form of an *application-program interface (API)* or a *framework*. To take decisions about deployment and reconfiguration, such a runtime has to be *configured* for the software application at hand, i.e., it has to be given a *deployment plan* capturing deployment mappings, constraints, and policies.

In principle, there are several possibilities to implement a deployment plan: (i) declaratively through a set of *configuration files*; (ii) through an *API/library* implemented in a general-purpose language; or (iii) through a dedicated DSL [37]. Naturally, these approaches have different trade-offs.

A framework/runtime with configuration files enforces declarativity at the configuration level, but its flexibility is limited, as options not accounted for at design time can hardly be injected through configuration files. On the other hand, a (well-designed) library can be flexible as the host language can be used to express peculiar configurations and is easier to adopt for users acquainted with the host language. However, its configuration may quickly become imperative and more challenging to maintain as complexity grows. Finally, a DSL [37] can be designed to be as expressive as needed, but it has a high maintenance cost (as the language maintenance stacks upon the library/API) and, potentially, a steep learning curve due to the need to learn a new (custom) language.

However, recent evolution in programming languages opened an additional strategy: *internal* DSLs, hybrids between libraries and standalone (external) DSLs [37]. Modern languages such as Kotlin, Groovy, Ruby, and Scala [38] provide specific syntactic features to enable the construction of APIs whose ergonomics is akin to the one of a dedicated language, but that are valid fragments in the host language. Although internal DSLs are de facto libraries in the host programming language (there is no clear boundary defining when a library becomes a DSL[1]), from a practical perspective they allow for great flexibility and ergonomics (although not total as the one of a custom DSL, as they are subject to the syntactic constraints of the host language) while retaining a reduced maintenance cost (as the host language ecosystem and tooling can be reused directly) and a gentle learning curve compared to stand-alone DSLs (as the syntax will be largely familiar to the host language users). Unlike external ones, internal DSLs allow the designer to directly fall back to the host language to implement peculiar entities or processes that the DSL cannot express. The approach has thus been successfully applied in several domains, from build systems[2] and ontologies [39] to hardware design [40] to logic [41] and aggregate programming [42].

[1] https://archive.is/wip/xAeiX.
[2] https://archive.is/5xtaN.

5.3. The system DSL : Components and required capabilities

The *System DSL* captures the concepts of *device type*, *component*, *capability*, and *requirement*. A reference sample is presented in Listing 1. The DSL provides simple means to define capabilities as types in the host language (Lines 1–3) and associate them with pulverised components for each device type (Lines 6–15). This DSL is meant to define constraints that the application imposes on the infrastructure, ruling out configurations that cannot support the system (for instance, deploying the sensor component of a device in a host that exposes no sensor). All combinations of components and capabilities defined as supported become amenable instead as *targets* of deployment or reconfiguration—forming what we may call the *deployment range*. In the example, e.g., component `Behavior` in `iot-sensor` is defined as executable on any host exposing `HighCPU` and/or `EmbeddedDevice` as capabilities (Line 12): it implies that the component can be dynamically moved to any host offering any such capability.

```
1  object HighCPU : Capability
2  object LowLatencyComm : Capability
3  object EmbeddedDevice : Capability
4
5  val conf = pulverizedSystem {
6    device("controller") {
7      Behavior and State deployableOn HighCPU
8      Communication deployableOn LowLatencyComm
9      Sensors deployableOn EmbeddedDevice
10   }
11   device("iot-sensor") {
12     Behavior deployableOn setOf(HighCPU, EmbeddedDevice)
13     Communication deployableOn LowLatencyComm
14     Sensors and Actuators deployableOn EmbeddedDevice
15   }
16 }
```

Listing 1: Example of *System DSL* usage. This code snippet defines three capabilities and a logical system composed of two device types, whose components are bound to the set of capabilities they need to execute.

5.4. The deployment DSL : Deployment domain, mapping, and reconfiguration

The *Deployment DSL* supports the definition of the mapping between pulverised components and specific hosts and the definition of reactive reconfiguration policies. It works with the concepts of `Host` (associated with `Capability`), `ReconfigurationEvent`, and `ReconfigurationRule`. The main goal of the *Deployment DSL* is to configure the *deployment unit*, which represents the smallest deployable unit of the pulverised system. It is characterised by a set of *pulverised components*, the *host* on which they are deployed, and the *logical device* they represent. Using Listing 2 as a reference example, we show how reconfiguration events (Lines 1–11) can be captured at the type-system level by expressing them as a predicate over an asynchronous flow of `events` influencing the system's state. Similarly, hosts can be configured by defining their names and capabilities (Lines 12–21). Once the hosts composing the system have been configured, the entire infrastructure can be set up (Lines 22–38) by defining, for each logical device, which host should host each pulverised component (Lines 26–29). Additionally, reconfiguration rules (Lines 30–37) can be defined to specify how the system should react to reconfiguration events. In this context, the DSL exposes a special syntax to express the migration of a pulverised component on a different host. Crucially, being the DSL internal, specialised behaviour not supported by the DSL can be defined in the host language directly: configuration blocks surrounded by curly

brackets are, in fact, plain lambda expressions [43] supporting any kind of computation.

```
// Reconfiguration events
expect fun cpuLoad(): Flow<Double>
expect fun batteryLevel(): Flow<Double>
object HighLoad : ReconfigurationEvent<Double>() {
  override val predicate = { it > 0.90 }
  override val events = cpuLoad()
}
object LowBattery : ReconfigurationEvent<Double>() {
  override val predicate = { it < 0.20 }
  override val events = batteryLevel()
}
// Available hosts
object Smartphone : Host {
  override val hostname = "smartphone"
  override val capabilities = setOf(EmbeddedDevice)
}
object Server : Host {
  override val hostname = "amazon-aws"
  override val capabilities =
        setOf(HighCPU, LowLatencyComm)
}
// Runtime setup and runtime reconfiguration rules
val infrastructure = setOf(Smartphone, Server)
val conf = ... // see Listing 1
pulverizationRuntime(conf,"iot-sensor",infrastructure){
  DeviceBehavior() startsOn Server
  DeviceCommunication() startsOn Server
  DeviceSensors() startsOn Smartphone
  DeviceActuators() startsOn Smartphone
  reconfigurationRules {
    onDevice {
      HighLoad reconfigures {
        Behavior movesTo Smartphone
      }
      LowBattery reconfigures { Behavior movesTo Server }
    }
  }
}
pulverizationRuntime(conf,"controller",infrastructure){
  /* ... */
}
```

Listing 2: Example of the usage of the domain-specific language for the runtime configuration. The example defines the runtime configuration of the system, specifying the initial deployment and the reconfiguration rules.

5.5. Implementation details and the runtime system

Several factors influenced the decision of the target technology for PulvReAKt. From the language perspective, we favoured statically typed languages, which often come with better contextual assist, and may help reducing the learning curve (we thus discarded Python, Javascript, and Ruby upfront). For the same reason, we decided to focus on the top-50 languages available on the well-known TIOBE index[3] to avoid selecting a language known only to a small niche. We then verified whether the language provided first-class support to the definition of DSLs, and, finally, we considered the possibility to target multiple runtimes.

Targeting multiple runtimes, and, in particular, native code, is paramount for the approach to be effective in a highly heterogeneous context such as the ECC. In fact, the framework should ideally be able to execute even on resource-constrained devices, and be able to support as much as possible the deployment of arbitrarily-written software components (namely, the interoperability with other languages should be as complete as possible).

We ultimately implemented the proposed DSL in Kotlin which, at the time of writing, targets the Java Virtual Machine (JVM), JavaScript, and native code for many platforms,[4] including mobile and wearable devices Scala could have been a valid alternative, but we deemed the multiplatform support of Kotlin a better fit for our needs. The framework has been open-sourced and released with a permissive licence.[5]

The runtime bases its operation on three main components: the `Spawner`, which is responsible for executing the components; the `UnitReconfigurator`, which is responsible for reconfiguring the *deployment unit* according to the `ReconfigurationEvent`; and the `Communicators`, which are used to establish the components' intra-communication. To start, the `PulverisationRuntime` needs to know the name of the *logical device*, the *host* on which it is running, and the configurations produced by the two DSLs. At the start of the system, using the provided configuration, the runtime spawns all the components that must execute on the *deployment unit*. Similarly, the `UnitReconfigurator` starts observing all the configured `ReconfigurationEvents` and registers the listeners for incoming reconfigurations. Then, each spawned component starts executing its logic and sharing data with other components via the `Communicator`, which is responsible for the components' intra-communication by leveraging the communication protocol. Finally, when a reconfiguration occurs, the `UnitReconfigurator` considers the new deployment configuration and, by interacting with the `Spawner`, starts or stops components in order to achieve the new setup.

At startup, the framework starts observing the `Reconfiguration Events` specified by the DSL, and, for each one, it collects the values produced by the phenomena observed by the event. These values are represented in the Kotlin language as `Flows`, namely asynchronous streams of values. Whenever a new value is produced by the `Flow`, the framework checks if the value satisfies the condition of the `ReconfigurationEvent`. Upon satisfaction of the conditions, the designated *deployment unit* responsible for that particular event initiates the reconfiguration process. The updated system configuration resulting from the reconfiguration process is then propagated to the other *deployment units* within the system.

If any of the conditions are satisfied, the specific *deployment unit* triggers the reconfiguration, and the new system configuration is propagated to the other *deployment units*.

To implement the migration of a component execution, each *deployment unit* is equipped with all the component defined by the DSLs, independently by the initial configuration. Whenever a reconfiguration is required, there is the problem of moving the execution of a component from one host to another. To do so, when a reconfiguration event is triggered, the component on the current host is stopped via the `UnitReconfigurator` and the `Spawner`. Subsequently, the `UnitReconfigurator` on the destination host – the intended migration target – interacts with the `Spawner` to start the component. This approach effectively simulates the migration of the component between hosts by managing the start and stop processes on the respective hosts.

During the application's startup phase, the framework conducts a validation check on the reconfiguration rules. This check aims to prevent the system from executing with invalid rules, such as attempting to migrate a component to a host lacking the required capabilities (as specified by the component). This validation ensures that the system

[3] https://archive.is/GiRce.

[4] https://archive.is/JSiTg.

[5] https://github.com/pulvreakt/pulvreakt.

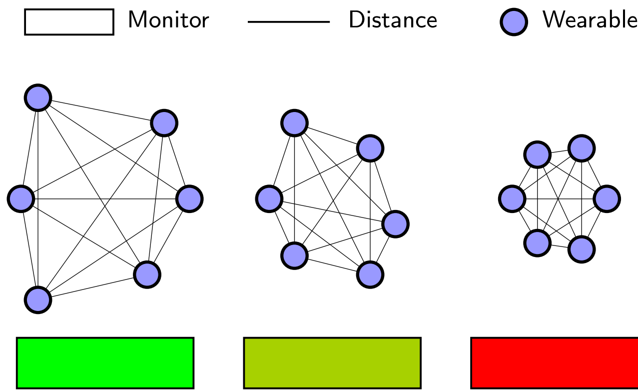


Fig. 5. Representation of the scenario described in Section 6. graph dots represent wearables, and arc lengths represent the distance between them. The rectangle at the bottom represents the monitor. The closer the wearables are, the higher the alert level (reddish tones).

consistently maintains a valid state, and that the reconfiguration rules remain applicable throughout runtime.

The configuration produced by the DSL needs a platform that instructs the system on how actual communication must be performed. In our current implementation, we provide out-of-the-box a JVM platform based on the AMQP protocol leveraging RabbitMQ, supporting both intra- and inter-device communication. The platform also optimises pulverised components of the same logical device ending up running on the same host: their communication happens directly by sharing memory without the need to go through (de)serialisation processes and networking. For the large-scale experiment, we implemented a second platform, this time to comply with the requirements of the simulation engine we used. The implementation required little effort and has been made available as part of the evaluation repositories.

6. Practical demonstrator: Crowd sensor

As a basic practical proof-of-concept demonstrator, we exercise the framework on a real-world testbed located in a laboratory. In particular, we want to avoid situations where too many people are too close to each other in the same room for safety reasons. To do so, we equip each person with a wearable device that can detect the distance from other similar devices via Bluetooth. For the sake of simplicity, we relied on smartphones to emulate wearables, and we used the Bluetooth received signal strength indicator (RSSI) to estimate the distance d between devices as:

$$d = 10^{\frac{R_{ref}-R}{10\cdot n}}$$

where R_{ref} is the RSSI reference value at 1 meter, R is the currently measured RSSI, and n is the path loss exponent, a parameter indicating the rate at which the RSSI decreases with distance. R_{ref} and n are environment-sensitive parameters that require calibration; we set up a test with two smartphones and measured, for our setup, $R_{ref} - 60$ dBm and $n = 2.0$ dB. We used a monitor to issue the warning by setting its colour from green to red depending on the average distance among people: the shorter the distance, the redder the tone. Fig. 5 shows a representation of the scenario.

The system is composed of two kinds of devices: *wearable* and *alarm*. Wearables consist of the following pulverised components: sensors (σ^w), perceiving other devices by their signal strength; behaviour (β^w), converting raw data from the sensor into messages for the alarm; and communication (χ^w) sending data to the alarm. They do not need a state or actuators. The alarm consists of the following pulverised components: β^a, computing the mean distance among devices; κ^a, storing the distances measured by the device for comparison with the recent past; χ^a, responsible for receiving the data from the wearable devices; α^a, enacting the alarm by showing a colour on the screen. It needs no sensors.

We use the following hosts in the scenario:

- multiple *Smartphones*, one per user, emulating wearables and used as distance sensors;
- a *Monitor* used to show the alarm;
- a *Server* used to run crowd estimation and wearable logic.

Initially, β^w is allocated to the *Server* while σ^w and χ^w are executed on the *Smartphones*, and all the *alarm* components are executed in the *Monitor*. In the experiment, we define a rule by which, if the *Server* is overloaded, β^w is moved to the *Smartphones* (with each smartphone running the β^w corresponding to its σ^w). During the experiment, we launch several heavy-duty processes on the *Server* and observe how the system reacts to the overload. We rely on the MQTT platform module for all the communication, meaning that the same protocol and broker handle both intra- and inter-device communication.

As soon as the reconfiguration event is triggered, the β component of the *Wearable* is moved from the *Server* to the *Smartphone* s hosting σ^w_s, relieving the server and preserving while retaining the system in nominal conditions. The reconfiguration of the system is entirely managed by the framework, which handles all the machinery and communication to move the component from one host to another.

This example has been used as a real-world testbed driving our experimental implementation. We do not report the entire configuration in this manuscript for brevity, but it is available, fully open-sourced, in the companion artefact [13], for those willing to inspect or exercise the whole system. To simplify testing, we also provide, in the same artefact, a simulation of the system where every device is containerised.

7. Evaluation

In this section, we perform an evaluation of the proposed framework on a larger-scale scenario via simulation. First, we set our evaluation goals (Section 7.1), connecting them with the research questions introduced in Section 2. Then, we simulate a city-scale distributed application deployed for a urban event and show how the dynamic reconfiguration may help to balance cost, energy consumption, and quality of service (QoS) (Section 7.2). At the end of the section, we discuss applicability (Section 7.3) and threats to validity (Section 7.4). The implementation of the scenario has been performed by interfacing PulvReAKt with the Alchemist Simulator [44] through a dedicated platform module. For inspectability and reproducibility, the experiments have been released with a permissive open-source licence[6] and archived for future reference on Zenodo [12].

7.1. Evaluation goals

We aim to evaluate PulvReAKt both in terms of *expressiveness* and *performance* of the resulting system. Specifically, we show how the framework can be used to develop a scalable pulverised system in a declarative and deployment-independent fashion, answering RQ1. Then, we demonstrate how the novel reconfiguration capabilities can be exploited to achieve better trade-offs between performance and resource utilisation compared to pre-defined static deployments, thus addressing RQ2.

7.2. Large scale urban collective computation

To show the potential benefit of automatic reconfiguration in CASs, we consider a city-scale collective computation in a urban setting.

[6] https://github.com/nicolasfara/experiments-2024-fgcs-pulverization-local-reconfiguration.

Suppose a large number of people are attending an event in a city to participate in a computationally intensive collective activity, e.g., a tournament of a geo-located game similar to Ingress[7] or Pokémon Go.[8]

Overall, the system features three types of networked devices: wearables (e.g., smartwatches), smartphones, and the cloud. Each wearable device is paired with a smartphone; which are connected to the cloud via 5G cellular network. We assume the application implementing the collective activity to be partitioned as prescripted by the pulverisation model; with two components capable to migrate on different devices: the application behaviour can execute either on smartphones or in the cloud; while the Global Positioning System (GPS) sensors used to determine each participant's position can be hosted either on the smartphones or on the wearables. We use this scenario as a reference to evaluate the effectiveness of PulvReAKt by opportunistically relocate these two components achieving previously unattainable trade-offs among energy consumption, cost to operate the system, and active participation of the users.

7.2.1. Metrics

We are interested in the following metrics:

- P_{system} (kW): the average power consumption of the system, computed as the sum of the power consumed by the devices and the cloud instances (under the assumption of a non-carbon-neutral energy mix, it can be used as a proxy for the carbon footprint);
- $\$_{cloud}$ (\$): the cost related to the cloud instances;
- Distance (km): the total distance walked by the participants, which we use as an indicator of the quality of experience (QoE) (the higher, the better), as users temporarily quit the event (and thus stop walking from the point of view of the application) when recharging; and
- h (minutes): the average time spent by users with their devices connected to a charging station instead of actively participating in the game, which we use as an indicator of the QoE (the lower, the better).

7.2.2. Energy model

We assume that the power consumption of the device is linearly dependent on the CPU usage, and we thus decided to estimate the energy per instruction (EPI) [45] of a typical CPU for a smartphone (EPI_{device}), a wearable device ($EPI_{wearable}$), and a server (EPI_{cloud}).

To do so, we take the thermal design power (TDP) of a Qualcomm Snapdragon 888 (5 W) and an Intel Xeon Platinum P-8124 (220 W), and we divide it by the score the CPU obtained in the popular Passmark benchmark[9] (9362 and 22674, respectively) obtaining an indication of the power per benchmark point ($^{5\ W}/_{9362\ pts}$ and $^{220\ W}/_{22\,674\ pts}$ respectively). We then used these ratios to estimate the relative EPI of the two CPUs, obtaining an EPI_{ratio} close to 1:18. Thus, we take the EPI estimation for the server CPU from [45], (approximately 100 nJ per instruction, considering 36 logical cores of our reference processor) and we assume the EPI of the smartphone CPU to be 18 times lower. Since no score is available for the wearable CPU, we assume the same efficiency of the smartphone CPU (as, in the case of smartwatches, the CPU is often based on a similar architecture) and a TDP five times lower (estimated from the specifications of a Google Pixel Watch 2). Once the EPIs of the platforms are known, we estimate the number of instructions for the pulverised components as follows: we assume the application to be a heavy-duty game, capable of draining the battery of a smartphone (approximately 4500 mAh in capacity, and thus, at 3.3 V, 14.85 Wh, or equivalently, 53 460 J) in 6 h of continuous usage, which divided by the EPI of the smartphone CPU provides the number of instructions. For the wearables, we assume that the GPS component can drain the battery in 6 h of continuous usage,[10] and we estimate the number of instructions accordingly. Additionally, we account for a variable quota of instructions that captures other activities of the devices (screen, operating system, other applications, etc.). Table 1 summarises the battery duration of the mobile and wearable devices when different components are allocated to them. For each device, we generate a variable OS workload, to make the battery discharge more realistic. This power consumption is computed by randomly selecting a percentage of the maximum power impact of the OS (as reported in Table 1) every simulated second.

Table 1
Battery duration of the wearable and mobile devices with different components allocation. With OS we indicate the operating system and other applications running on the device. The reported time represents the average duration of the battery assuming operational devices for which the components allocation is constant.

Device	Allocated Comp.	Avg. drain time
Smartphone	$\beta + \sigma + \chi + \text{OS}$	6h
	$\beta + \chi + \text{OS}$	10h
	OS	24h
Wearable	$\sigma + \chi + \text{OS}$	6h
	OS	24h

Table 2
Simulation parameters.

Simulation parameter	Values
PoIs	15
β Offloading threshold	$\Updownarrow_x \ \lvert x \in \{0, 10, 20, 30, 40, 100\}$
σ Offloading policies	Smartphone, hybrid, wearable
Device count	300
Random seed	1, 2, …, 1000

7.2.3. Cost model

We assume the cloud to be composed of a set of Amazon *AWS* instances, precisely, *m5.16xlarge* instances (using our reference server CPU). We consider for the *AWS* (Elastic Compute Cloud) an hourly cost of \$3.584.[11] We estimate the number of instances required to run the pulverised components by dividing the total power consumption of the cloud by the TDP of the server, assuming the CPU to be the dominant energy consumption component, and assuming that each vCPU on AWS maps to a logical core of the underlying hardware. The specific parameters of the cloud instances are determined based on the CPU specifications, in order to have the closest match to the reference server CPU described in Section 7.2.2.

7.2.4. Experimental setup

We assume the event starts with users displaced at random positions in the city, and their smartphones and wearables devices charge is randomly initialised ensuring at least 60% of battery, and a probability of the 10% that a device requires to be recharged when joining the event. The game requires users to physically move around the city, reaching PoIs and performing some operations there (e.g., play a collaborative game or visit an attraction). We assume users with low battery to turn off the application and recharge the device before rejoining the game. Fig. 6 shows a snapshot of the simulation.

We configure the framework to move the behaviour to the cloud when the battery level (measured as a percentage of the maximum charge) is below a threshold λ, and to move it back to the device when the battery is fully charged.

The $\Updownarrow_x$ symbol represents the different *behaviour reconfiguration strategies*, where x is the battery charge percentage threshold at which

[7] https://www.ingress.com/.
[8] https://pokemongolive.com/.
[9] https://www.cpubenchmark.net/.
[10] https://archive.is/2JEwe.
[11] https://archive.is/HhBLI.

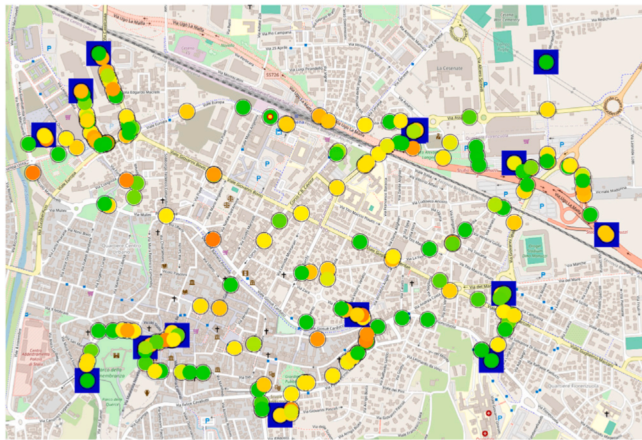


Fig. 6. A snapshot of the large-scale simulation. Smartphones are represented as coloured circles, with the colour indicating the average battery level between smartphone and wearable (green when charged, red when low). The blue squares represent the PoIs.

the β component is moved to the cloud; the special cases $\Updownarrow_0$ and $\Updownarrow_{100}$, represent, respectively, smartphone-only and cloud-only configurations. The σ component reconfigurations are instead: **smartphone**, where only the smartphone's GPS is used; **wearable**, where only the wearable's GPS is used; or **hybrid**, where the GPS is allocated to the device with the highest battery level, with the reconfiguration rule triggered every 5% of battery discharge of the device the component is currently allocated on.

In each experiment, the initial battery levels, the initial position of the users, and the decision of the next PoI to visit is stochastic. As a consequence, we repeat each experiment 1000 times with different random seeds. Table 2 summarises the simulation parameters. The results and errors shown in the remainder of this paper are computed over these repetitions.

7.2.5. Results

In the following, we show the results of the simulations based on the metrics defined in Section 7.2.

Travelled distance. Fig. 7 reports the average distance walked by the participants in the last 30 min over time, comparing different sensor (one per chart) and behaviour (one line per chart) reconfiguration strategies. As expected, $\Updownarrow_0$ requires more frequent recharges, and the need for recharging is less frequent as it gets more likely for the behaviour workload to be offloaded to the cloud, reaching the best results with $\Updownarrow_{100}$. This condition is evident but when the sensor allocation strategy is to always keep the GPS sensor on the wearable: in this case, wearables' discharge dominate over smartphones, and the overall QoE flattens regardless of the behaviour allocation strategy. By looking at the behaviour of the system towards the end of the experiment, (the rightmost part of the charts showing hybrid and smartphone sensor allocation strategies), we can see that the *hybrid* strategy outperforms the other ones, as, regardless the behaviour allocation strategy, devices tend to not undergo a second recharge cycle (also, the first recharge cycle is delayed). In the *hybrid* strategy, alternation between of the GPS allocation between smartphones and the wearables discharges batteries more uniformly, extending the overall battery life and impacting positively on the QoE.

Time on recharge. Since for this application the time spent recharging the device is time not spent using the application actively, we investigate such metric as a proxy for the QoE; results are depicted in Fig. 8. Data shows that the wearable-only sensor allocation strategy is the worst across the board (as wearable tend to discharge faster), except in the case of the $\Updownarrow_0$ configuration, in which hosting both behaviour and sensors on the smartphones accelerates the discharge so much that it becomes the dominating factor. Notably, the *hybrid* strategy performs visibly better than the others when no other compensation strategy is in place, namely, in $\Updownarrow_0$ and $\Updownarrow_{100}$ configurations. Otherwise, the behaviour induced by the balance between the smartphone and wearable battery life is less impactful, as the cloud offload can compensate similarly for the battery discharge (but, as we will show later, at a higher cost).

Power consumption. We compute the overall power consumption, combining smartphones, wearables, and cloud instances, applying the energy model introduced in Section 7.2.2. The results are depicted in Fig. 9. The power consumption gives us an indication of the overall system power impact. This metric, assuming the same non-carbon-neutral energy mix is used to operate the cloud and recharge the mobile devices, can also be used as a proxy metric to evaluate the relative carbon footprint of *operating* the system. As expected, power consumption is dominated by the cloud instances, thus, in cases in which the behaviour is never ($\Updownarrow_0$) or always ($\Updownarrow_{100}$) offloaded to the cloud, a different sensor allocation strategy does not impact cost. In experiments where the behaviour can be relocated, purely cost-wise, the always-on-wearable strategy seems to perform best, but it does so at the expense of QoE, as many devices discharge their wearable well before the smartphone battery has been consumed enough to reach the cloud relocation threshold. The behaviour of the smartphone-only and hybrid strategies is similar, and comparing the two is interesting, as for low reconfiguration thresholds ($\Updownarrow_{10}$ and $\Updownarrow_{20}$), it outperforms the smartphone-only strategy, while induces more power consumption as the threshold increases ($\Updownarrow_{30}$ and $\Updownarrow_{40}$). This result is motivated by the fact that the hybrid strategy tends to keep the sensor on the wearables when the battery discharge of the smartphone is higher, and less when it is lower: the larger the fraction of battery at which the smartphone delegates its behaviour to the cloud, the more the sensor reconfiguration can extend the duration of the participation in the activity, the higher the cost. In other words, the sensor relocation strategy is capable to extend the period of time in which the behaviour is offloaded to the cloud longer than the period in which is bound to the smartphone, thus causing a higher power consumption (and a better QoE). Although the interaction among multiple reconfiguration rules on power consumption can be non-trivial, data shows that it can be a very effective way of balancing power consumption and QoE.

Cloud cost. The last metric we are interested in is the monetary cost associated to cloud usage. In Fig. 10, we show the cost of the cloud for each reconfiguration threshold and sensor allocation strategy. Unsurprisingly, these are very similar to the power consumption results, with the addition of a constant amount of money that is spent to always keep at least one instance online for the application to work, as the communication component in the implementation under analysis is cloud-allocated. Thus, very similar considerations apply, including the effect of the hybrid strategy in extending the period of time in cloud (and thus whose cost grows faster with the cloud offload threshold compared with the smartphone-only strategy).

7.2.6. Final considerations

We have provided a comprehensive evaluation of the proposed framework, showing that thanks to the runtime reconfiguration we can achieve trade-offs between the cloud cost, the distance walked by the participants, and the time spent on recharge, otherwise not possible with a static deployment (answering the RQ2 and RQ1). The results show that appropriate reconfiguration strategies can be beneficial to extract the most out of complex configurations in which multiple devices can play the same role (for instance, because sensors are replicated). In this sense, we have shown that opportunistically moving the sensor execution between the smartphone and the wearable device can extend the battery life of the devices, reducing the time spent on recharge and thus maximising the QoE. Finally, we have provided a cloud cost analysis highlighting how, with a dynamic reconfiguration, good performance for the system under study could be achieved with less than half of the cost of a cloud-only deployment, thus enabling novel trade-offs unavailable without relocation.

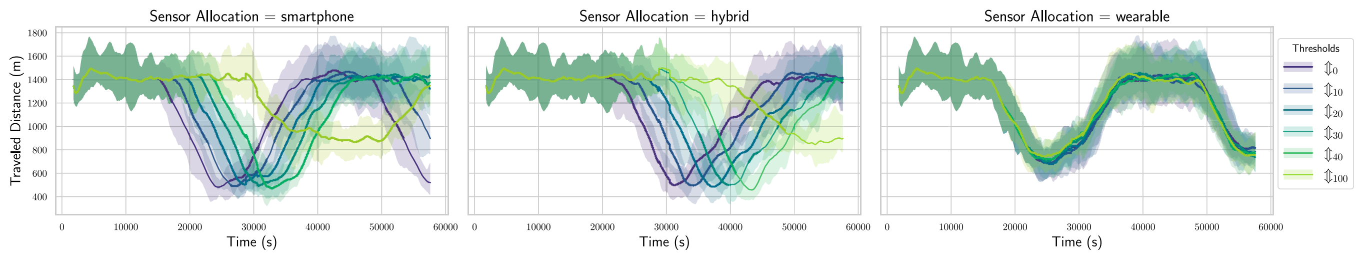


Fig. 7. Average distance walked by the participants in the last 30 min. Coloured lines represent different reconfiguration thresholds. One chart is produced for every sensor allocation strategy: smartphone-only (left), hybrid (center), wearable-only (right). Shades show $\pm 1\sigma$.

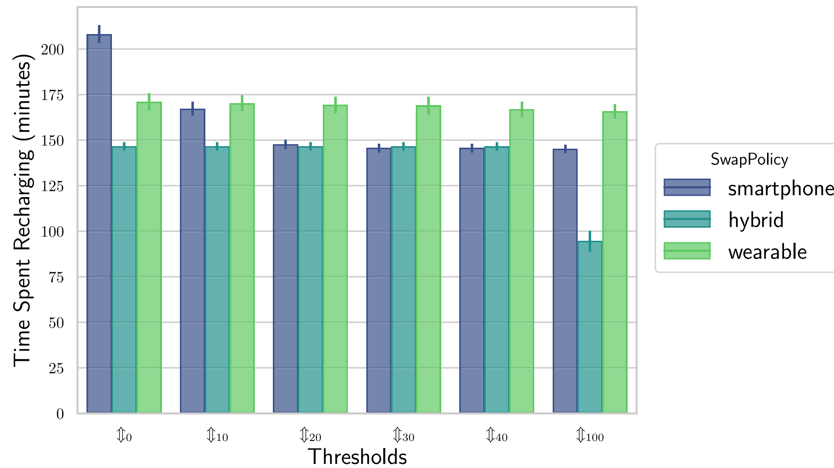


Fig. 8. Average time spent by the participants to recharge their devices. For each reconfiguration threshold, the three sensor allocation strategies are shown. The vertical lines over the bars show $\pm 1\sigma$.

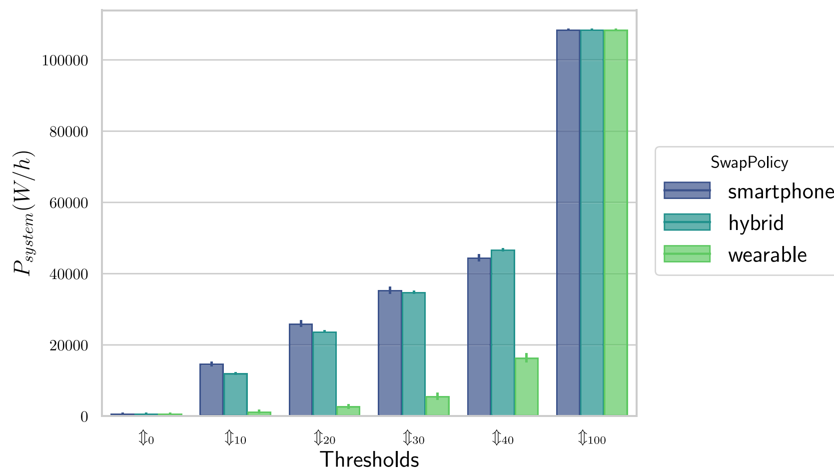


Fig. 9. Overall system power consumption. For each reconfiguration threshold, the three sensor allocation strategies are shown. The vertical lines over the bars show $\pm 1\sigma$.

7.3. Applicability

PulvReAKt can be profitably applied to support flexible deployment of any system that can be conveniently modelled according to the pulverisation approach (cf. Section 3.2), which means, broadly speaking, a collective of devices equipped with sensors (and possibly actuators) able to compute and interact with neighbours—i.e. systems found in scenarios like the IoT, edge computing, swarm robotics, and the like.

Therefore, PulvReAKt offers a versatile solution for a wide range of applicative scenarios and infrastructures. It is designed to address complex deployments in emergent infrastructures such as the cloud–edge continuum, as well as systems and environments of highly dynamic nature (cf. mobility, failure, varying loads).

As showed in Sections 5.4 and 5.3, the main advantage offered is the possibility to specify the model of the system in terms of classes of devices in the system, then provide different deployment strategies according to the target infrastructure, without touching the business logic of the system. In this way, the same application can be deployed on many infrastructures with minimal effort, and reducing all the risks related to different deployment configurations for the same application, strategy aligned with the approach proposed in [11].

Conversely, the opposite strategy can be adopted: for the same infrastructure specification, several system specifications can be provided. In this workflow, the focus is put on relying on the same infrastructure for deploying different systems reusing the same deployment specification, useful to test the same application in

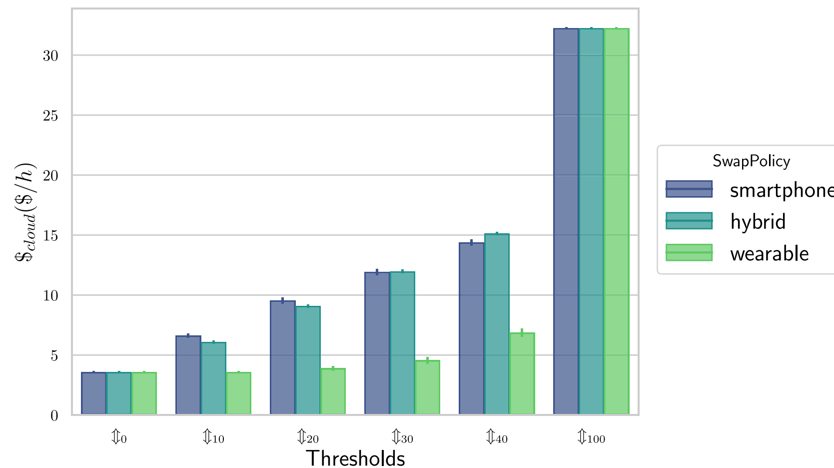


Fig. 10. Cloud cost for each reconfiguration threshold and sensor allocation strategy. The vertical lines over the bars show $\pm 1\sigma$.

different deployments, such as via simulation as showed in Section 7.

The proposed framework allows you to specify more complex systems and infrastructures, compared to the examples proposed in this paper. To do this simply specify the right components for each type of device that takes part in the system, and specify the infrastructure structure and then leave the rest of the work to the framework. With this approach, the framework can scale from simple systems to real-world complex deployments.

PulvReAKt can be used to design and deploy *heterogeneous* systems characterised by heterogeneity of system's devices (cf. thin and thick hosts), dynamic changing requirements, and a strong interaction with the physical world. Therefore, we argue that PulvReAKt can be adopted for managing the reconfiguration of collective systems in the context of the IoT and more generally for large-scale cyber–physical systems (CPSs), as showed by the results in Section 7.2.

7.4. Threats to validity

Our experiments consider a scenario where the application is pulverised and the behaviour can be executed either on the device or in the cloud, depending on the battery level. To do so, we assume the same kind of smartphones to be used by all the participants, and we assume the same battery consumption for each of them. Similarly, we suppose the cloud to be composed of a single server type. In this regard, we believe that the consumption model we used is a good approximation of the real-world scenario (see Section 7.2), but we acknowledge that the results may vary if different devices are used.

Another aspect that may affect the results is the "external load" of both the cloud and the devices: the cloud is likely used for other purposes, and the smartphones may run in the background other services and/or applications. In the experiment, we have emulated the external load by increasing the overall host load with a random coefficient. Although this is an approximation, we believe that does not compromise the validity of the results. This issue can be alleviated by using real-world data, but this is out of the scope of this work.

Finally, we evaluated four reconfiguration strategies, doing so by varying the thresholds of each strategy. Despite the limited number of tested configurations, we believe that the results are representative of the potential of the framework.

8. Conclusion and future work

In this work, we presented a practical DSL and framework for the runtime reconfiguration of pulverised CAS applications. The original idea of pulverisation was to define applications considering arbitrary networks of logical devices, decompose (pulverise) these logical devices into small deployment units, and then define their desired deployment at a later time. The specialised DSL introduced in this work shows that this idea can be realised in practice. Moreover, we extended the original idea of pulverisation with the possibility of specifying runtime reconfiguration policies, which can be leveraged to achieve better performance, cost, and resource usage trade-offs (RQ1). Finally, we provide a real-world demonstrator for the technology and show via simulation that the reconfiguration approach can scale better and provide benefits compared to a static approach (RQ2).

In its current form, the proposed DSL has two limitations that will be addressed in the future. First and foremost, it currently does not consider openness, as hosts get specified in the runtime and reconfiguration DSLs as a closed set. The second limitation is related to the policies that can be expressed in the reconfiguration DSL: currently, only policies that can be assessed at the local host level can be assessed. A policy that requires considering the state of multiple hosts would require resorting to the host language's primitive mechanisms and make explicit communications outside the framework [46], thus breaking the abstraction. This limitation can be addressed by extending the current reconfiguration block of the DSL so that queries can be performed on any reachable host of the system, thus making all communications (including reconfiguration-related ones) pass through the channels controlled by the framework. Concurrently with these extensions, we intend to add support for additional heterogeneity, including device types and communication networks, to extend the applicability to a larger number of practical scenarios. Finally, a future research direction is a practical evaluation of the framework in a large-scale real-world application to better learn how the intrinsic complexity of a real deployment (latencies, lost packets, failing sensors, etc.) can impact the reconfiguration rules and the overall system performance.

CRediT authorship contribution statement

Nicolas Farabegoli: Writing – original draft, Validation, Software, Data curation, Conceptualization. **Danilo Pianini:** Writing – review & editing, Writing – original draft, Data curation, Conceptualization. **Roberto Casadei:** Writing – review & editing, Writing – original draft, Conceptualization. **Mirko Viroli:** Writing – review & editing, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

https://doi.org/10.5281/zenodo.11401482.

Acknowledgements

The work was partially supported by the Italian PRIN project "CommonWears" (2020HCWWLP).

References

[1] S.S. Gill, M. Xu, C. Ottaviani, P. Patros, R. Bahsoon, A. Shaghaghi, M. Golec, V. Stankovski, H. Wu, A. Abraham, M. Singh, H. Mehta, S.K. Ghosh, T. Baker, A.K. Parlikad, H. Lutfiyya, S.S. Kanhere, R. Sakellariou, S. Dustdar, O.F. Rana, I. Brandic, S. Uhlig, AI for next generation computing: Emerging trends and future directions, Internet Things 19 (2022) 100514, http://dx.doi.org/10.1016/j.iot.2022.100514.

[2] L.F. Bittencourt, R. Immich, R. Sakellariou, N.L.S. da Fonseca, E.R.M. Madeira, M. Curado, L. Villas, L.A. DaSilva, C. Lee, O.F. Rana, The internet of things, fog and cloud continuum: Integration and challenges, Internet Things 3–4 (2018) 134–155, http://dx.doi.org/10.1016/j.iot.2018.09.005.

[3] Z. Hong, W. Chen, H. Huang, S. Guo, Z. Zheng, Multi-hop cooperative computation offloading for industrial IoT-edge-cloud computing environments, IEEE Trans. Parallel Distrib. Syst. 30 (12) (2019) 2759–2774, http://dx.doi.org/10.1109/TPDS.2019.2926979.

[4] H. Wang, H. Zhao, J. Zhang, D. Ma, J. Li, J. Wei, Survey on unmanned aerial vehicle networks: A cyber physical system perspective, IEEE Commun. Surv. Tutor. 22 (2) (2020) 1027–1070, http://dx.doi.org/10.1109/COMST.2019.2962207.

[5] M. Afrin, J. Jin, A. Rahman, A. Rahman, J. Wan, E. Hossain, Resource allocation and service provisioning in multi-agent cloud robotics: A comprehensive survey, IEEE Commun. Surv. Tutor. 23 (2) (2021) 842–870, http://dx.doi.org/10.1109/COMST.2021.3061435.

[6] R. Casadei, G. Fortino, D. Pianini, A. Placuzzi, C. Savaglio, M. Viroli, A methodology and simulation-based toolchain for estimating deployment performance of smart collective services at the edge, IEEE Internet Things J. 9 (20) (2022) 20136–20148, http://dx.doi.org/10.1109/JIOT.2022.3172470.

[7] R. Casadei, Artificial collective intelligence engineering: A survey of concepts and perspectives, Artif. Life 29 (4) (2023) 433–467, http://dx.doi.org/10.1162/artl_a_00408.

[8] J. Spolsky, The law of leaky abstractions, in: Joel on Software, A Press, 2004, pp. 197–202, http://dx.doi.org/10.1007/978-1-4302-0753-5_26.

[9] J. Liu, E. Ahmed, M. Shiraz, A. Gani, R. Buyya, A. Qureshi, Application partitioning algorithms in mobile cloud computing: Taxonomy, review and future directions, J. Netw. Comput. Appl. 48 (2015) 99–117, http://dx.doi.org/10.1016/j.jnca.2014.09.009.

[10] J. Arcangeli, R. Boujbel, S. Leriche, Automatic deployment of distributed software systems: Definitions and state of the art, J. Syst. Softw. 103 (2015) 198–218, http://dx.doi.org/10.1016/j.jss.2015.01.040.

[11] R. Casadei, D. Pianini, A. Placuzzi, M. Viroli, D. Weyns, Pulverization in cyber-physical systems: Engineering the self-organizing logic separated from deployment, Future Internet 12 (11) (2020) 203, http://dx.doi.org/10.3390/fi12110203.

[12] N. Farabegoli, Nicolasfara/experiments-2024-fgcs-pulverization-local-reconfiguration: 1.11.6, 2024, http://dx.doi.org/10.5281/zenodo.11401482.

[13] N. Farabegoli, Nicolasfara/pulvreakt-crowd-room: 1.2.2, 2024, http://dx.doi.org/10.5281/zenodo.10637846.

[14] P. Kruchten, The 4+1 view model of architecture, IEEE Softw. 12 (6) (1995) 42–50, http://dx.doi.org/10.1109/52.469759.

[15] A. Carzaniga, A. Fuggetta, R.S. Hall, D. Heimbigner, A. Van Der Hoek, A.L. Wolf, A Characterization Framework for Software Deployment Technologies, Technical Report, Colorado State University, 1998.

[16] A. Gazis, E. Katsiri, Middleware 101, Commun. ACM 65 (9) (2022) 38–42, http://dx.doi.org/10.1145/3546958.

[17] J. Beal, D. Pianini, M. Viroli, Aggregate programming for the internet of things, IEEE Comput. 48 (9) (2015) 22–30, http://dx.doi.org/10.1109/MC.2015.261.

[18] M. Viroli, J. Beal, F. Damiani, G. Audrito, R. Casadei, D. Pianini, From distributed coordination to field calculus and aggregate computing, J. Log. Algebr. Methods Program. 109 (2019) 100486, http://dx.doi.org/10.1016/j.jlamp.2019.100486.

[19] T. Vale, I. Crnkovic, E.S. de Almeida, P.A. da Mota Silveira Neto, Y.C. Cavalcanti, S.R.d. Meira, Twenty-eight years of component-based software engineering, J. Syst. Softw. 111 (2016) 128–148, http://dx.doi.org/10.1016/j.jss.2015.09.019.

[20] I. Crnkovic, S. Sentilles, A. Vulgarakis, M.R.V. Chaudron, A classification framework for software component models, IEEE Trans. Softw. Eng. 37 (5) (2011) 593–615, http://dx.doi.org/10.1109/TSE.2010.83.

[21] N. Medvidovic, R.N. Taylor, A classification and comparison framework for software architecture description languages, IEEE Trans. Softw. Eng. 26 (1) (2000) 70–93, http://dx.doi.org/10.1109/32.825767.

[22] A.L. Lemos, F. Daniel, B. Benatallah, Web service composition: A survey of techniques and tools, ACM Comput. Surv. 48 (3) (2016) 33:1–33:41, http://dx.doi.org/10.1145/2831270.

[23] R. Casadei, Macroprogramming: Concepts, state of the art, and opportunities of macroscopic behaviour modelling, ACM Comput. Surv. (2023) http://dx.doi.org/10.1145/3579353.

[24] P. Weisenburger, J. Wirth, G. Salvaneschi, A survey of multitier programming, ACM Comput. Surv. 53 (4) (2021) 81:1–81:35, http://dx.doi.org/10.1145/3397495.

[25] P. Ghosh, H. Tu, T. Krentz, G. Karsai, S.M. Lukic, An automated deployment and testing framework for resilient distributed smart grid applications, in: COINS, IEEE, 2022, pp. 1–6, http://dx.doi.org/10.1109/COINS54846.2022.9854934.

[26] A.P. Achilleos, K. Kritikos, A. Rossini, G.M. Kapitsaki, J. Domaschka, M. Orzechowski, D. Seybold, F. Griesinger, N. Nikolov, D. Romero, G.A. Papadopoulos, The cloud application modelling and execution language, J. Cloud Comput. 8 (2019) 20, http://dx.doi.org/10.1186/s13677-019-0138-7.

[27] R. Boujbel, S. Rottenberg, S. Leriche, C. Taconet, J. Arcangeli, C. Lecocq, MuScADeL: A deployment DSL based on a multiscale characterization framework, in: COMPSAC Workshops, IEEE Computer Society, 2014, pp. 708–715, http://dx.doi.org/10.1109/COMPSACW.2014.120.

[28] H. Coullon, L. Henrio, F. Loulergue, S. Robillard, Component-based distributed software reconfiguration:A verification-oriented survey, ACM Comput. Surv. 56 (1) (2023) http://dx.doi.org/10.1145/3595376.

[29] R.E. Ballouli, S. Bensalem, M. Bozga, J. Sifakis, Four exercises in programming dynamic reconfigurable systems: Methodology and solution in DR-BIP, in: Leveraging Applications of Formal Methods, Verification and Validation. ISoLA, Proceedings, Part III, in: Lecture Notes in Computer Science, Vol. 11246, Springer, 2018, pp. 304–320, http://dx.doi.org/10.1007/978-3-030-03424-5_20.

[30] R.D. Nicola, A. Maggi, J. Sifakis, The dream framework for dynamic reconfigurable architecture modelling: theory and applications, Int. J. Softw. Tools Technol. Transf. 22 (4) (2020) 437–455, http://dx.doi.org/10.1007/s10009-020-00555-2.

[31] M.B. Chhetri, H.P. Luong, A.V. Uzunov, Q.B. Vo, R. Kowalczyk, S. Nepal, I. Rajapakse, ADSL: An embedded domain-specific language for constraint-based distributed self-management, in: ASWEC, IEEE Computer Society, 2018, pp. 101–110, http://dx.doi.org/10.1109/ASWEC.2018.00022.

[32] A. Dearle, G.N.C. Kirby, A.J. McCarthy, A framework for constraint-based deployment and autonomic management of distributed applications, in: 1st International Conference on Autonomic Computing (ICAC 2004), 17-19 May 2004, New York, NY, USA, IEEE Computer Society, 2004, pp. 300–301, http://dx.doi.org/10.1109/ICAC.2004.3.

[33] J.A. Hewson, P. Anderson, A.D. Gordon, Constraint-based autonomic reconfiguration, in: SASO, IEEE Computer Society, 2013, pp. 101–110, http://dx.doi.org/10.1109/SASO.2013.23.

[34] F. Alvares, É. Rutten, L. Seinturier, A domain-specific language for the control of self-adaptive component-based architecture, J. Syst. Softw. 130 (2017) 94–112, http://dx.doi.org/10.1016/j.jss.2017.01.030.

[35] H. Song, R. Dautov, N. Ferry, A. Solberg, F. Fleurey, Model-based fleet deployment in the IoT-edge-cloud continuum, Softw. Syst. Model. 21 (5) (2022) 1931–1956, http://dx.doi.org/10.1007/s10270-022-01006-z.

[36] M. Villari, M. Fazio, S. Dustdar, O. Rana, D.N. Jha, R. Ranjan, Osmosis: The osmotic computing platform for microelements in the cloud, edge, and internet of things, IEEE Comput. 52 (8) (2019) 14–26, http://dx.doi.org/10.1109/MC.2018.2888767.

[37] M. Voelter, S. Benz, C. Dietrich, B. Engelmann, M. Helander, L.C.L. Kats, E. Visser, G. Wachsmuth, DSL Engineering - Designing, Implementing and Using Domain-Specific Languages, dslbook.org, 2013, URL http://www.dslbook.org.

[38] P. Riti, Practical Scala DSLs: Real-World Applications Using Domain Specific Languages, A Press, Berkeley, CA, 2018, http://dx.doi.org/10.1007/978-1-4842-3036-7.

[39] J.P. Balhoff, Scowl: a scala DSL for programming with the OWL API, J. Open Source Softw. 1 (1) (2016) 23, http://dx.doi.org/10.21105/joss.00023.

[40] F. Serre, M. Püschel, DSL-based hardware generation with scala: Example fast Fourier transforms and sorting networks, ACM Trans. Reconfigurable Technol. Syst. 13 (1) (2020) 1:1–1:23, http://dx.doi.org/10.1145/3359754.

[41] G. Ciatto, R. Calegari, A. Omicini, 2P-KT: A logic-based ecosystem for symbolic AI, SoftwareX 16 (2021) 100817:1–100817:7, http://dx.doi.org/10.1016/j.softx.2021.100817.

[42] R. Casadei, M. Viroli, G. Aguzzi, D. Pianini, ScaFi: A scala DSL and toolkit for aggregate programming, SoftwareX 20 (2022) 101248, http://dx.doi.org/10.1016/j.softx.2022.101248.

[43] J. Järvi, J. Freeman, C++ lambda expressions and closures, Sci. Comput. Program. 75 (9) (2010) 762–772, http://dx.doi.org/10.1016/J.SCICO.2009.04.003.

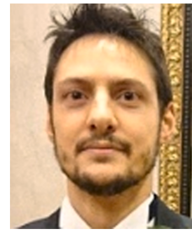
[44] D. Pianini, S. Montagna, M. Viroli, Chemical-oriented simulation of computational systems with ALCHEMIST, J. Simul. 7 (3) (2013) 202–215, http://dx.doi.org/10.1057/jos.2012.27.

[45] Y.S. Shao, D.M. Brooks, Energy characterization and instruction-level energy model of intel's xeon phi processor, in: ISLPED, IEEE, 2013, pp. 389–394, http://dx.doi.org/10.1109/ISLPED.2013.6629328.

[46] N. Farabegoli, D. Pianini, R. Casadei, M. Viroli, Dynamic iot deployment reconfiguration: A global-level self-organisation approach, 2024, http://dx.doi.org/10.2139/ssrn.4798700.

Nicolas Farabegoli is a Ph.D. student at Alma Mater Studiorum–Università di Bologna (Italy). He graduated in Computer Science at the University of Bologna in 2023, with a thesis on the design and development of a framework for flexible deployments in cloud–edge systems. His research interests include Cloud–edge computing, large-scale distributed systems, collective adaptive systems, and aggregate programming.

Danilo Pianini is senior assistant professor at Alma Mater Studiorum—Università di Bologna (Italy). His research is focused on self-organising systems, complex systems engineering, and simulation, topics on which he published more than 70 papers in international journals and conferences. He made dozens of contributions to the open-source community, and he is the chief architect and lead engineer of the open-source Alchemist Simulator and Protelis aggregate programming language. He served as PC chair of IEEE ACSOS 2021 and as PC member of multiple IEEE and ACM conferences including AAMAS, ACSOS, COORDINATION, ICAART, and IJCAI.



Roberto Casadei is an assistant professor at Alma Mater Studiorum–Università di Bologna (Italy). His research revolves around software engineering and distributed artificial intelligence. He has 60+ publications in international journals and conferences on topics including collective intelligence, aggregate computing, self-* systems, and IoT/CPS. He also leads the development of the open-source ScaFi aggregate programming toolkit. He has been serving in the OC/PC of multiple conferences such as ACSOS, COORDINATION, ICCCI, and SAC, and as editorial board member of JAISCR.

Mirko Viroli is Full Professor in Computer Engineering at the University of Bologna, Italy. He is an expert in foundations of computer science and programming, object-oriented programming, advanced software development, software engineering and self-adaptive/self-organising pervasive computing systems. He is author of more than 300 papers, of which more than 80 on international journals. His GoogleScholar h-index is 49 with >8500. He is member of the Editorial Board of IEEE Software magazine, and was program chair of the ACM Symposium on Applied Computing (SAC 2008 and 2009), and IEEE Self-Adaptive and Self-Organizing systems (SASO 2014) conferences.