A Methodology and Simulation-Based Toolchain for Estimating Deployment Performance of Smart Collective Services at the Edge

Roberto Casadei, *Member, IEEE*, Giancarlo Fortino, *Fellow, IEEE*, Danilo Pianini, *Member, IEEE*, Andrea Placuzzi, Claudio Savaglio, *Member, IEEE*, and Mirko Viroli, *Senior Member, IEEE*

***Abstract*—Research trends are pushing artificial intelligence (AI) across the Internet of Things (IoT)–edge–fog–cloud continuum to enable effective data analytics, decision making, as well as the efficient use of resources for QoS targets. Approaches for collective adaptive systems (CASs) engineering, such as aggregate computing, provide declarative programming models and tools for dealing with the uncertainty and the complexity that may arise from scale, heterogeneity, and dynamicity. Crucially, aggregate computing architecture allows for "pulverization": applications can be decomposed into many deployable micromodules that can be spread across the ICT infrastructure, thus allowing multiple potential deployment configurations for the same application logic. This article studies the deployment architecture of aggregate-based edge services and its implications in terms of performance and cost. The goal is to provide methodological guidelines and a model-based toolchain for the generation and simulation-based evaluation of potential deployments. First, we address this subject methodologically by proposing an approach based on deployment code generators and a simulation phase whose obtained solutions are assessed with respect to their performance and costs. We then tailor this approach to aggregate computing applications deployed onto an IoT–edge–fog–cloud infrastructure, and we develop a corresponding toolchain based on Protelis and EdgeCloudSim. Finally, we evaluate the approach and tools through a case study of edge multimedia streaming, where the edge ecosystem exhibits intelligence by self-organizing into clusters to promote load balancing in large-scale dynamic settings.**

***Index Terms*—Cloud services, collective services, cyber–physical systems, deployment methodology, edge intelligence, mobile and ubiquitous systems, pulverizable architectures, service middleware and platform, simulation.**

I. Introduction

EDGE computing (EC) is a complementary paradigm to cloud computing that provides computational resources and intelligence at the edge of the network, close to Internet of Things (IoT) devices and users. So, it benefits communication latency and data rate and supports scalability through decentralization and locality. Enriching the edge with artificial intelligence (AI) capabilities [1]–[3] can be vital to unlocking the potential of the IoT, enabling large-scale data processing and reactivity in decision making. However, edge ecosystems tend to be complex due to the heterogeneity of participating devices and the high dynamicity of relationships and goals (as induced, e.g., by mobility, failure, environmental changes, and user activity). Prominent issues include defining efficient edge structures [4], coordinating edge resource providers and consumers [5], and supporting decision making for reconfiguration and load balancing [6].

Recently, the aggregate paradigm has proved valuable as an approach for engineering *opportunistic services* in IoT and EC scenarios [7]–[9] and for programming *collective edge intelligence* [3], [6]. A key benefit of aggregate computing is that its architecture allows *pulverizing* (i.e., finely partitioning) applications (which we call *aggregate applications* or *aggregate systems*) into several logical components and deployment units [10]. These units can be spread to available infrastructure for defining a particular *deployment configuration*. This flexibility in deployment is critical to fully exploiting the IoT-edge-fog-cloud infrastructural continuum opportunities.

This work focuses on predicting how the *deployment* affects the *performance* and *costs* of smart edge services expressed as aggregate applications. This is a significant issue since suboptimal deployments can negatively affect performance (e.g., system reactivity to change due to latencies, or unavailability caused by energy depletion) and costs (e.g., in terms of network capacity or energy consumption) associated with these services. Therefore, for an effective engineering process of complex systems [11], it is crucial to evaluate and compare different target deployment configurations to mitigate the risk of ineffective deployments and reconfigurations (which may cause further costs and temporary QoS reduction). This is a problem of *methods* and *tools*, which should guide and support engineers across the various engineering phases. Therefore, this article delineates a methodology and presents tools for assessing aggregate application deployments through simulation. Most specifically, we provide the following contributions.

1) We propose a *methodology* applicable to *pulverizable* (partitionable) *systems*, which leverages deployment

Manuscript received 2 March 2022; accepted 28 April 2022. Date of publication 4 May 2022; date of current version 7 October 2022. This work was supported in part by the Italian Ministero dell'Istruzione, dell'Università e della Ricerca (MIUR) PRIN 2017 Project "Fluidware" under Grant 2017KRC7KT, and in part by the EU/MUR FSE REACT-EU PON R&I 2014–2022 under Grant CCI2014IT16M2OP005. *(Corresponding author: Danilo Pianini.)*

Roberto Casadei, Danilo Pianini, Andrea Placuzzi, and Mirko Viroli are with the Department of Computer Science and Engineering (DISI), Alma Mater Studiorum–Università di Bologna, 47522 Cesena, Italy (e-mail: danilo.pianini@unibo.it).

Giancarlo Fortino is with the Department of Informatics, Modeling, Electronics and Systems, Università della Calabria, 87036 Rende, Italy.

Claudio Savaglio is with the Institute for High Performance Computing and Networking (ICAR), National Research Council, 87036 Rende, Italy.

Digital Object Identifier 10.1109/JIOT.2022.3172470

generators and simulators for assessing performance and costs associated with a set of target deployment configurations.
2) We implement the methodology specifically for aggregate computing applications by mapping aggregate specifications to possible deployment configurations in a way reminiscent of Infrastructure as Code (IaC) [12].
3) We develop a toolchain, integrating the *Protelis* aggregate programming engine [13] with the *EdgeCloudSim* simulator [14] for measuring metrics related to edge–cloud deployments.
4) We apply the approach and toolchain to the edge multimedia streaming case study presented in [6], which represents a large-scale scenario where edge intelligence is exploited for balancing the load on edge servers through dynamic clustering.

The remainder of the article is organized as follows. In Section II, we report an account of state-of-the-art approaches for engineering and simulation of collective adaptive systems (CASs) and provide background on the pulverizable architecture of aggregate computing applications. In Section III, we present our methodology and toolchain. In Section IV, we provide a quantitative evaluation of the approach. Finally, Section V provides conclusive thoughts and discusses significant directions for future work.

II. Background and Related Work

A. Collective Adaptive Systems Development

In this section, we review the state of the art on CAS engineering, which aims to analyze and design the emergent behavior of large-scale situated cyber–physical systems. Example applications include crowd engineering for safe navigation and dispersal [15], smart mobility [16], situated problem solving [9], trust and reputation systems [17], robotics [18], [19], and resilient management of ICT infrastructures [20]. A recent survey of models, methods, and tools for rigorous CAS engineering can be found in [21]. In the following, we review *programming* and *system specification* approaches supporting the development of CAS-based applications.

Techniques for CAS programming generally leverage one or more of the following abstractions: *ensembles*, namely, dynamic collections of devices; *collective communication interfaces*, namely, abstractions enabling individual components to communicate with groups; and *field-like data structures*, namely, mechanisms to address data belonging to an entire group of components. Other approaches inspiring CAS development techniques can be found among *macroprogramming* and *spatial computing* contributions, as surveyed in [22]. In the following, we examine an approach to CAS programming based on the *computational field* abstraction, deriving from the spatial computing and coordination tradition [23].

1) Aggregate Computing: Aggregate computing [15] is a full-fledged paradigm for CAS engineering. It formally founds on field-based coordination [23] to compositionally express collective adaptive behavior from a global perspective. Field-based coordination is captured by field calculi [23] and implemented through languages, such as the standalone Protelis [13] and the Scala-internal ScaFi [24]. In field calculi, the whole behavior of the system is specified in terms of expressions manipulating *computational fields*, namely, maps from system components to values. So, for instance, querying a temperature sensor across a network yields a field of floating-point values denoting temperatures, checking whether the mean temperature across the neighbor's contributions exceeds a threshold yields a field of Boolean values denoting potentially critical conditions, and so on.

From an operational perspective, each component (or device) belonging to an aggregate system has some middleware support for collaborating to the aggregate application, which consists of an aggregate program plus configuration related to connectivity and scheduling. From a logical point of view, each device operates in *computational rounds*, each of which is composed of the following steps.
1) *Context Acquisition:* Sensor information, messages from neighbors, and state information are collected.
2) *Computation:* The aggregate program is applied against the local context, yielding an output value and an outbound coordination message.
3) *Actuation:* The output value can be used to drive actuators.
4) *Coordination:* The outbound coordination message is expected to be broadcasted to all the neighbors of the device.

Rounds of different devices may be asynchronous. This execution protocol is independent of the concrete aggregate program: different programs generate different outputs, and, consequently, different messages for neighbors. The more complex the program, the larger are the messages to be exchanged with neighbors. Note that though the aggregate program is the same for all the devices, the individual behaviors would generally be different, as it would be evaluated against a different context.

The complete coverage of the theory and practice of aggregate computing is beyond the scope of this article: the interested reader can refer to [23] for more details on the approach and the main formal properties. For the sake of this article and to provide a clue about how the aggregate paradigm works, let us consider a simple but paradigmatic example: the *self-healing channel*. This is a distributed, dynamic structure in a network that provides a hop-by-hop path from a *source* device to a *target* device, represented as a Boolean field mapping the devices comprised in the channel to `true` and the devices outside the channel to `false`. This channel can be programmed by *composing* other functions leveraging *gradients* [25] to estimate 1) the distance from any node to the source; 2) the distance from any node to the target; and 3) the source-to-target distance. With these three pieces of information available locally, each device can determine whether it belongs to the channel or not by exploiting simple geometry (the triangle inequality). Moreover, the above functions can be *reused* for other algorithms as well. Crucially, regarding the dynamics, note that as devices move in the network, the fields from which the channel is computed self-adjust, eventually converging to their correct value, and the same happens for the channel as well. We remark that such a "global" computation (a channel spanning a network) is executed in a fully decentralized way, in terms of local sensing (distance to neighbors) and communication (as prescribed by the aggregate program). The reusability of aggregate behaviors, modeled as functions from input to output fields, enables the paradigm to scale with complexity and to discover high-level coordination patterns [6]—one of these, implementing clustered feedback loops, is especially convenient for edge coordination and considered for the case study in Section IV.

2) Pulverizable Architecture of Aggregate Applications: The execution and coordination protocol described in the previous section is not a rigid schema: it is flexible and may be adapted, both offline and online, to take into account various tradeoffs between performance and cost. For instance, the frequency of operation of the devices can be adjusted to closely match the rate of change of the phenomena under monitoring. Moreover, the responsibilities of a component of the aggregate (sensing, computation, actuation, and coordination) do not need to be deployed together. This notion, known as *pulverization* [10], enables the partitioning [26] of aggregate applications into several deployment units, which may be deployed variously on a target ICT infrastructure, leading to potentially different performance and costs [8], [10].

Fig. 1 summarizes the notion of pulverization for an aggregate computing system. In this partitioning model, the responsibilities corresponding to the activity of an individual logical device are split into the following parts (Fig. 1).

1) *Knowledge:* This part keeps track of the software state, which is also instrumental for coordinating the other parts.
2) *Sensor:* This part is responsible for extracting data from the local context/environment. Sensed data are stored in the device state.
3) *Actuator:* This part is responsible for acting on the local context/environment. Actuation is driven by the data stored in the device state.
4) *Behavior:* This part is responsible for processing the local context as represented in the device state to determine how the state should evolve, what actuations need to be carried out, and what data are to be exchanged with neighbors.
5) *Communication:* This part is responsible for exchanging information with neighbors, as prescribed by the behavior.

Then, an implementation would provide a different deployable component (or service) for each of these parts, and a deployment would specify which components are put onto which physical devices (Fig. 1).

In aggregate computing, the aggregate system that is programmed is, first, a logical entity, potentially decoupled from an underlying physical system. Typically, a logical node (Fig. 1) is associated with a concrete node of a networked system (e.g., a robot, a smartphone, or a server in a cluster)—its *physical twin*—and provides a "reasoning cycle" to turn its sensor observations to actuation instructions (cf. Fig. 1). However, certain responsibilities (e.g., computation, state, and communication) can be, in principle, offloaded to *other* physical devices—e.g., edge, fog, or cloud nodes (cf. Fig. 1). These responsibilities may be offloaded because of limitations of the physical twin (e.g., it is a *thin host* with little storage, energy, and computational capabilities) to enable neighboring relationships that are not related to physical connectivity, or in the context of a strategy to optimize the execution of the aggregate computation [8], [10] (e.g., exploiting co-location

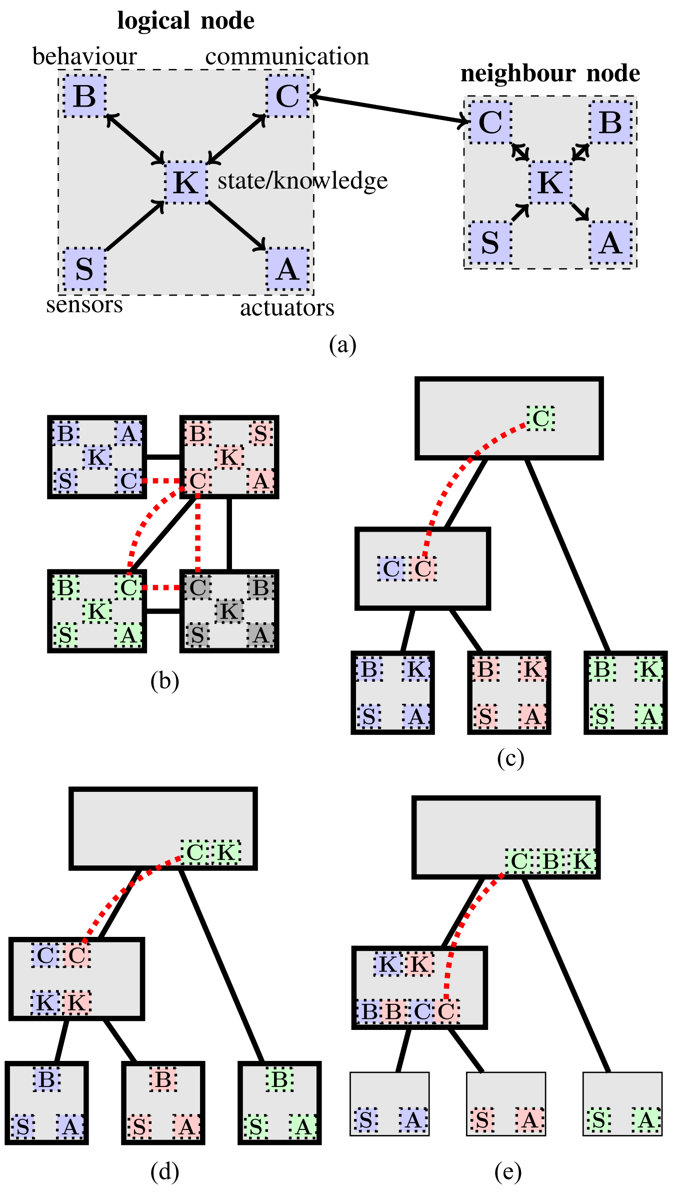

Fig. 1. Pulverizable architecture in aggregate computing: device responsibility destruction and notable deployments (Figures adapted from [10]). Notation: solid-border boxes denote hosts; bold-border boxes denote *thick* hosts; normal-border boxes denote *thin* hosts; solid edges denote connections between hosts; dashed-border boxes denote logical components, with letters indicating their function (behavior, communication, knowledge, sensors, and actuators); differently colored boxes denote (components of) different logical nodes; red dashed edges denote logical connections, i.e., neighboring relationships (not shown for co-located components). (a) Logical device, split into subcomponents, and one of its neighbors. (b) Peer-to-peer architecture: no offloading, 1-to-1 mapping between logical and physical devices. (c) Communication component offloaded. (d) Communication and state components offloaded. (e) Full offloading: IoT hosts can be thin.

of the communication components to realize zero-latency and zero-bandwidth interaction).

In the following sections, we show how pulverizable architectures can be considered in the context of a methodology-supported engineering process and evaluate performance and costs associated with different aggregate application deployments (Section IV).

B. Simulators for Collective Adaptive Systems on the Edge

Assessing how a collective adaptive application will perform before its actual deployment is paramount yet somewhat tricky: the system it applies to comprises multiple and possible heterogeneous entities; as such, most classic testing techniques do not apply straightforwardly; moreover, the correctness (or accuracy) of the system is usually measured with multiple continuous metrics, which need to be evaluated for many different situations. Moreover, many variables in real-life applications come from experience and are most often estimations. Simulation can be used robustly in these cases, as solutions can be picked based not just on their outcome with the expected value of the parameters, but considering the reliability of the behavior across a wide range of situations. Consequently, simulation has a prominent role in the overall system engineering procedure. Since even *functional* requirements are generally defined in terms of global behavior, simulation enters the game early, as it is the leanest way to verify if prototypes work as expected. Simulators meant to support prototyping and development usually abstract away network-level and nonapplication-level details, prioritizing scalability and debugging of the system behavior. Typical simulators frequently leveraged in the software design phase include Repast [27], NetLogo [28], and Alchemist [29], with the latter featuring first-class support for aggregate computing specifications defined in Protelis [13] or ScaFi [30].

Once the system is functionally verified, information concerning low-level details, such as networking and power consumption need to be assessed: a solution that was found to provide the system with the required capabilities may be unsustainable for the actual deployment, or show degraded performance under some circumstances not captured by the abstract simulated model. In such a case, two options are open: searching for another implementation option at the application level or finding a more efficient mapping of the application logic on the deployed system (e.g., as proposed in [31]). Systems supporting pulverization offer a principled way to tackle the issue, generalizing the second option: the functional logic is separated from its deployment details and several configuration options can thus be explored.

Exploring these configurations requires dedicated tools that expose deployment details and accurately capture information regarding resource consumption. Traditionally, these tools include network simulators (such as OMNeT++ [32] and NS-3 [33]), operating-system-level simulators (such as TOSSIM [34]), and cloud-oriented simulators (e.g., CloudSim [35], DEVS [36], and Recap [37]). Recently, simulators specifically conceived to provide comparative analysis between the edge- and cloud-based deployments of an IoT system are gaining attention. These technologies were surveyed [38], [39] and classified with respect to the provided Quality/Metrics Characteristics of ISO/IEC 25010/25023 [40] (functional suitability, performance efficiency, compatibility, reliability, maintainability, portability, etc.) and modeling features (infrastructure, application, and resource management modeling, scalability, and mobility models).

From both surveys, EdgeCloudSim [14] emerges as one of the richest edge–cloud simulators. EdgeCloudSim is a Java-written application distributed with an open-source licence, based on CloudSim, and it has been exploited in dozens of works and different application scenarios for evaluating IoT applications performances, offloading strategies, resource allocation schemes, etc. The simulator models a layered architecture composed of *end devices*, *edge*, and *cloud*. The base simulated entity is the *task*, a process generated by end devices with a predetermined resource and network consumption that can be offloaded to upper layers (edge or cloud), through, respectively, local (WLAN) or wide-area networks (WAN). An *orchestrator* is in charge of implementing the rules and policies for handling incoming devices' tasks. EdgeCloudSim focuses on three primary performance metrics: 1) service time (by distinguishing between its two components, i.e., networking and computation time); 2) service failure rate (which can be due to networking-, mobility- or computation-related factors); 3) and resource utilization (in terms of CPU or bandwidth overloads). Simulations are parameterized by simulation time, device count, packet size, task length, and network bandwidth through a set of declarative XML specifications, respectively, devoted to the application, device, and scenario modeling—thus relieving the end user from the need to tinker with the simulator source code.

C. Other Deployment Methodologies

Several approaches dealing with the deployment of complex systems and featured by different degrees of comprehensiveness have been proposed so far. Table I shows a qualitative comparison of the main works by specifying their scope (general or special), target (particular objectives and supported deployment), contribution (i.e., from full-fledged methodologies to dedicated strategies and tools), and evaluation environment (namely, network simulators, numerical frameworks, or real testbeds). In particular, it can be immediately noticed that they mainly focus on the allocation/placement of generic resources, tasks, services, or applications by considering a double-layer infrastructure (cloud plus edge/fog) supporting the developers with a single contribution to be evaluated through a network simulator. Conversely, the methodology we propose and its associated toolchain are flexible enough to open a larger design space (IoT–edge–fog–cloud continuum), enabling the definition of the business logic of a distributed software independent of any deployment constraint or concern, assuming that the underlying software system can be pulverized. In this sense, the methodology introduces a cleaner separation of concerns; the main advantage with respect to the traditional methods is the possibility to defer the choice of which host will ultimately execute which part of the software.

TABLE I
QUALITATIVE COMPARISON OF DEPLOYMENT APPROACHES ACCORDING TO THEIR PURPOSE (GENERAL OR SPECIAL), GOAL, TARGET INFRASTRUCTURE (IOT, EDGE, FOG, CLOUD), CONTRIBUTIONS (METHODOLOGY, MODELS, (OPTIMAL) ALGORITHMS, STRATEGY, TOOL, TOOLCHAIN), AND EVALUATION ENVIRONMENT (SIMULATOR, TESTBED, NUMERICAL FRAMEWORK)

	Pur.	Goal	Target	Contrib.	Eval.
[41]	G	Service Placement	C+F	Me	Sim
[42]	S	Service Placement	F	Me	Test
[43]	G	Service Placement	C+F	Me	Sim
[44]	G	Resource Allocation	C+E	Me	Sim
[45]	G	Workload Allocation	C+E	OA	Num
[46]	S	DSP application Placement	C+E	S	Sim
[47]	S	Edge Device Location	E	OA	Sim
[48]	S	Task Allocation	C+E	S	Sim
[49]	G	Application Placement	C+E	Mo+OA	Sim
[50]	G	Resource Allocation	C+E	M, T	Sim+Test
[51]	S	Service Placement	C+E	A	Num
Ours	S	Application deployment	I+C+E	Me+Tc	Sim

III. METHODOLOGY

This section presents our contribution, which consists of a methodology for simulation-based deployment evaluation (Section III), and an implementation of the methodology to the aggregate computing paradigm, along with a prototype toolchain providing support to its various phases (Section III-B).

A. Methodology for Simulation-Based Deployment Evaluation for Pulverizable IoT-Edge-Fog-Cloud Systems

As discussed in Section II-A2, pulverized systems provide a clean separation between the application business logic and its actual deployment. This section discusses how this feature can be leveraged in a methodology, providing insights into how combinations of different pulverization would behave on different infrastructures The input information for the method has two elements:

1) functional requirements for the application;
2) possible target infrastructures.

Functional requirements must be satisfied by creating an appropriate specification in a pulverizable language or platform (in our prototype implementation, we relied on the aggregate programming language Protelis [13]), producing a partitioned application. The available potential target infrastructures need to be captured into a formal machine-readable model. This operation is similar to a "reverse" IaC [12]. In IaC, computation resources (servers, virtual machines, and their configuration) are managed and provisioned via machine-readable declarative configuration files; in our case, configuration files should be produced as descriptors of the possible infrastructural configurations. Crucially, in the case the final system's target infrastructure was instanced via IaC (which is likely for distributed systems using a modern DevOps automation pipeline), the IaC descriptor could easily play the role of infrastructure descriptor for the proposed methodology as well.

Information on the infrastructure model and the partitioned application is then provided to a deployment generator, a configurable software component that finds all possible valid deployments of pulverized components onto the possible infrastructures, generating the corresponding simulation files. Simulations are then executed, in our case, by relying on EdgeCloudSim, and performance analysis is performed. The methodology is summarized in Fig. 2.

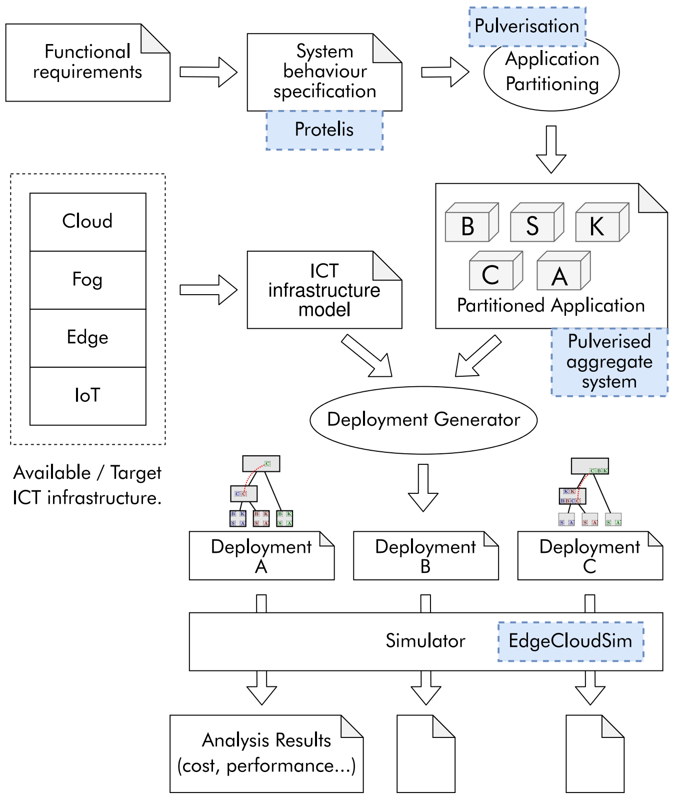

Fig. 2. Overview of the approach. (The blue boxes denote specific instantiations of the corresponding methodology elements that have been exercised in this article.)

Results can then be: 1) interpreted by the developers to gain insights on the most suitable strategy for the pulverization of the system; 2) used by the operations team to figure out which deployment allows meeting the requirements while saving resources; and 3) even integrated into the quality assurance automated pipeline. In this sense, the proposed methodology also contributes to the evolution of the best practices for the development of distributed systems by realizing a predeployment performance evaluation that could influence the IaC and deployment phases, thus allowing to block the deployment in case the system was found to have a relevant performance regression.

B. Application to Aggregate Computing

We have created a prototype implementation of the tooling required for applying the methodology [52]. In particular, we have selected the Protelis aggregate programming language as pulverizable behavior specification, and the EdgeCloudSim platform for simulating the deployed system, framed in blue-filled boxes in Fig. 2.

The deployment generator module is at the core of the approach, representing the novel element introduced in the toolchain. It works as an adapter between the high-level pulverized program specification and the network simulation tool, and has the following responsibilities.

1) Given a behavioral specification (in our case, a Protelis program), providing cost models compatible with the low-level simulator (in our case, EdgeCloudSim).
2) Given a set of possible infrastructures, filtering those compatible with the requirements of the pulverized system.
3) For all the plausible combinations between infrastructures and deployments of pulverized components, generating all the valid simulation configurations.
4) Analyzing all these configurations by exercising the network simulator.

1) Complexity Estimation of Pulverized Aggregate Programs: The first step of our analysis methodology requires estimating the resources that are required to run a pulverized system. In such directions, there exist several options that capture the specification at different levels of abstraction and provide differently grained estimates.

One approach is the static analysis [53], which takes as input the source code, the binaries it produces (if compiled), or some intermediate product of compilation; builds an internal model of the program; and then performs the analysis by searching for known patterns, without actually executing the code. Usually, static analysis tools are meant to intercept style inconsistencies, dodgy code snippets, bad practices, bugged patterns, and security vulnerabilities; however, the same technique can also be used to gain insights on the complexity and, especially by data-flow analysis (namely, static prediction of the possible runtime values of some variables), on some bounds of the size featuring the exchanged network messages. In our prototype implementation, we rely on static analysis to estimate the computation load required by Protelis programs, done by intercepting the intermediate representation of the abstract syntax tree produced by the Protelis interpreter before execution, and then by estimating the execution cost of each subtree. Since Protelis is higher-order [54], we had to take into account function references and lambda expressions, hijacking the standard interpretation machinery to explore their body. The peculiar Xtext-based implementation [55] of the language has been relevant for simplifying the process and for driving the choice of Protelis as target language for the analysis. On the other hand, the lack of a static type-checker hindered our data-flow analysis, so we expect other aggregate computing implementations (such as the Scala-internal DSL ScaFi [24]) to be amenable of a more detailed analysis (although at a higher implementation cost due to the complexity of the host language).

Another approach to evaluating the expected performance of code is (micro)benchmarking, which requires instrumented execution of the software and measures (rather than estimates, as it is done by static analysis) the cost of executing software. Although this kind of measurement sounds attractive; the measures may be affected by a very significant error and the outcomes could be much less precise than expected [56]. This is due to the inherent complexity of modern computers and software stacks: CPUs are equipped with several layers of caches that heavily impact performance, e.g., even comparing different list implementations by timing their use can generate astonishing results);[1] the operating system scheduling policies introduce additional variability; compiler tuning can produce, from the same source, executables with different performance; and, finally, language runtimes, such as the Java Virtual Machine or the Common Language Runtime introduce further layers of complexity due to internal caching, garbage collection, just-in-time (de)optimization [57], and other mechanisms. Even though, in principle, (micro)benchmarking is a viable option for a complexity estimator and should probably be part of an all-round tool, the vast number of variables to keep in check made it a technique unsuitable for a prototypical demonstrator as the one we are presenting for this work; thus, we used static analysis.

As mentioned in Section II-B, usually a simulation step is required during the design to understand whether the desired behavior is being achieved. These high-level simulation tools, although often not capturing enough of the low-level details, can be leveraged to extract valuable information on the system. For instance, the Alchemist Simulator [29] does run actual aggregate code in simulations. We indirectly exploited this capability by excerpting the small portion of code in charge of executing the aggregate program and emulating the delivery of messages to other devices, configuring it manually using the Protelis networking module, and interposing a serialiser in-between: this way, we estimated message sizes rather precisely. This is an instance of a more general practice where simulators used in the design step are leveraged for obtaining insights on the behavior of the system (in our case, the expected message size), and this information is then used in the detailed deployment evaluation.

2) Formal Deployment Specifications: As discussed in Section III-A, the proposed approach requires a formal specification of the actual target platform. The most likely *reification* for a full-fledged tooling implementation would likely be a translation directly made from the IaC definitions; however, there is no physical target in our prototypical proof of concept. For this reason and the sake of simplicity, we have developed our own lightweight syntax, rather than implementing an ICT infrastructure model translating existing tools' definitions into EdgeCloudSim-compatible environments descriptors. A relevant factor in our decision has been the lack, at the time of writing, of any widely accepted standard language for IaC. The arguably most widespread syntax is the custom language used by the Terraform tool [58], which would require a customized parser.

Our descriptor is thus a standard YAML[2] file capturing hardware, networking, and infrastructure parameters. An excerpt of such configuration is presented in Fig. 3. It is possible to specify multiple values for most keys: every combination of such values is then tested in simulation.

A second descriptor, exemplified in Fig. 4, defines the possible deployments of pulverized components. Omitted keys have a default target; in particular, the state component is intended to be deployed alongside the behavior component unless

[1]https://archive.is/BJAuq
[2]https://yaml.org/spec/1.2/spec.html

```
infrastructure:
  wanPropagationDelay: [0.15] # seconds
  gsmPropagationDelay: [0.16] # seconds
  lanInternalDelay: [0.01] # seconds
  wlanBandwidth: [200] # Megabits per second
  wanBandwidth: [15] # Megabits per second
  gsmBandwidth: [10] # Megabits per second
  numberOfHostOnCloudDatacenter: [1]
  numberOfVmOnCloudHost: [1]
  coreForCloudVm: [128]
  mipsForCloudVm: [120000] # MIPS
  ramForCloudVm: [128000] # Megabytes
  storageForCloudVm: [1000000] # Megabytes
  coreForMobileVm: [1]
  mipsForMobileVm: [1000] # MIPS
  ramForMobileVm: [1800] # Megabytes
  storageForMobileVm: [32000] # Megabytes
  edgeCount: 1
  edgeCores: 16
  edgeMips: 12000 # MIPS
  edgeRam: 16000 # Megabytes
  edgeStorage: 250000 # Megabytes
  # One of: RANDOM_FIT, FIRST_FIT, NEXT_FIT, WORST_FIT,
  # BEST_FIT, or ONLY_EDGE
  orchestratorPolicies: NEXT_FIT
```

Fig. 3. Infrastructure descriptor.

```
pulverised_system:
  sensingUploadData: 0.5 # Kilobytes
  actuatorDownloadData: 1.0 # Kilobytes
  behaviour:
    name: "behaviour"
    # Probabilities in percent
    probCloudSelection: [ 0.0, 25.0, 50.0, 75.0, 100.0 ]
    poissonInterarrival: [ 5.0 ] # Seconds
    taskLength: # Computed via static analysis
      from_protelis_module: scr.pt
  communication:
    name: "communication"
    # Probabilities in percent
    probCloudSelection: [ 0.0, 25.0, 50.0, 75.0, 100.0 ]
    poissonInterarrival: [ 5.0 ] # Seconds
    taskLength: [ 15.0 ] # Millions instructions
```

Fig. 4. Pulverized system descriptor.

otherwise specified, and sensors and actuators are intended as deployed on end devices.

3) Automated Simulation Execution and Data Analysis: Out tool performs three tasks in order to execute all the required simulations: a *generation task* that compiles the input descriptors and generates a set of EdgeCloudSim configuration files, an *execution task* that runs all simulations, and an *analysis task* producing graphs. The generation task:

1) performs the Cartesian product of hardware, networking, and infrastructure parameters;
2) estimates the Protelis program complexity and related message size;
3) generates EdgeCloudSim configuration files for each combination of device count, edge server count, and (if an interval is provided rather than a number) Protelis program complexity and message size; and
4) finally produces a descriptor for the execution task with instructions on all the combinations to be executed.

Once done, the execution task is in charge of launching multiple repetitions of the simulation. Finally, the analysis task executes on the produced results. The final part of the evaluation depends on the nonfunctional requirements of the application: unless a deployment configuration is strictly superior to another under all the metrics (which is pretty unlikely in real-world situations), identifying the best one requires factoring in the impact of any metric of interest. For instance, for a social real-time distributed game, latency could be vastly more important than data rate; on the contrary, if the application business logic concerns streaming prerecorded high-definition videos, the data rate is predominant over low latency. In our case, the analysis task has been used to produce the charts included in Section IV, along with many others that we have used to interpret better the system behavior that was not included for the sake of brevity (still, they are available on the same repository hosting the prototype [52]). We have written the former two tasks in Kotlin as part of the Gradle-based build automation tooling of the prototype. Instead, charts have been produced via a MATLAB script, executed via Octave through a GitHub Action.

IV. Evaluation

This section exercises our proposed methodology and toolchain by performing a deployment analysis for a case study from the literature. We have selected an existing implementation to show that the approach applies to existing code without any impact on the specification; most specifically, we have taken an implementation of the self-organizing coordination regions pattern [6] applied to the collection of local user-generated multimedia streaming. Our goal is to demonstrate that through the proposed methodology, a predeployment analysis can be executed provided a specification of the target infrastructure and the pulverized behavior. Section IV-A reports how the experiment is configured and Section IV-B presents evaluation results.

A. Configuration

The behavior of the simulated system is obtained from the experiments presented in [6]: we have our analyzer with the Protelis source code found in [59], obtaining an estimation of the millions of instructions required on average to compute the behavior and an estimate of the message size.

We then created the infrastructure descriptor, trying to map a reasonably realistic target platform. Our reference target comprises nine edge servers with an associated Wi-Fi access point that end devices can connect to. EdgeCloudSim models wireless data rate reduction due to environmental factors and shared resources internally, provided a *maximum* data rate. We selected such a maximum data rate by observing results in the literature for 802.11ac Wave-1-certified [60] and Wave-2-certified [61] devices (released in 2016). Although theoretical performance tables show much higher performance,[3] practical studies suggest a maximum data rate of 123 Mb/s for Wave-1 devices [60] and 282 Mb/s for Wave-2 devices [61]. We decided to pick an intermediate value of 200 Mb/s as our maximum expected data rate. Edge servers are equipped with 16 GB of RAM and are configured to mirror an existing dedicated edge server processor (Intel Xeon D[4] model D-1540)[5] featuring eight cores, 16 parallel threads, a base frequency of 2.00 GHz, and a thermal design power of 45 W. Edge servers can communicate with the cloud via a WAN connection with

[3]https://mcsindex.net/
[4]https://archive.is/x8owH
[5]https://archive.is/JUoIG

TABLE II
AVAILABLE CHOICES FOR THE DEPLOYMENT OF PULVERIZED COMPONENTS IN DIFFERENT DEPLOYMENT CONFIGURATIONS. ● INDICATES THAT THE COMPONENT CAN BE DEPLOYED ON THE LOCAL DEVICE DIRECTLY, ▦ THAT IT CAN GET DEPLOYED ON EDGE, AND ☁ ON CLOUD. ACTUATOR AND SENSOR COMPONENTS ARE ALWAYS DEPLOYED ON THE LOCAL DEVICE. THE STATE COMPONENT IS ALWAYS DEPLOYED ON THE SAME HOST OF THE BEHAVIOR COMPONENT. IN $B \downdownarrows C$, BEHAVIOR AND COMMUNICATION COMPONENTS ARE FORCED TO BE HOSTED ON THE SAME HOST, IN $B \rightleftarrows C$ THEY CAN BE LOCATED ON DIFFERENT HOSTS, AND IN C ONLY THE COMMUNICATION COMPONENT IS HOSTED ON EITHER EDGE OR CLOUD

Deployment	Sensors-Act.s	Behaviour and State	Comm.
$B \rightleftarrows C$	●	▦ or ☁	▦ or ☁
$B \downdownarrows C$	●	▦ or ☁	▦ or ☁
C	●	●	▦ or ☁

a stable data rate of 15 Mb/s, while end devices can do the same via LTE at 10 Mb/s. We have simulated a typical modern (at the time of writing) cloud server system, featured by 128 GB of RAM and an AMD EPYC[6] 7702[7] processor with 64 cores, 128 parallel threads, a base clock of 2.00GHz and a thermal design power of 200W. This configuration is affine to large cloud instances available at the time of writing by several well-known cloud service providers. We have tested with three work modes for the infrastructure.

1) *1-tier:* Only edge devices are enabled, the cloud is unreachable.
2) *2-tier:* Both edge devices and cloud are reachable, pulverized components of end devices are tied to run on their closest edge server.
3) *2-tier With Edge Orchestrator:* Same as 2-tier, but pulverized components may run on any edge server.

In such an infrastructure, we have tested three different pulverized deployments, which are visually summarized in Fig. 5.

1) C, where the end devices host almost all computation (they are *thick* nodes), only delegating the communication component to the cloud or the edge.
2) $B \rightleftarrows C$, where devices only host sensors and actuators (they are *thin* nodes), delegating the rest to the edge or the cloud.
3) $B \downdownarrows C$, similar to $B \rightleftarrows C$ but with the additional constraint that behavior and communication components are forced to be located on the same host.

In every case, components that are deployable (see Table II for the available options) on edge and cloud have a probability $P_{☁}$ of being hosted on the cloud instead of on edge. We consider the following cases as *baselines*, as they could be realized traditionally, without pulverization.

1) *C With $P_{☁} = 0$:* An application designed from the beginning to run on the end devices and communicate through the edge.
2) *C With $P_{☁} = 1$:* An application designed from the beginning to run on the end devices and communicate through cloud-mediated messages.

[6]https://archive.is/g7jN9
[7]https://archive.is/M0ChF

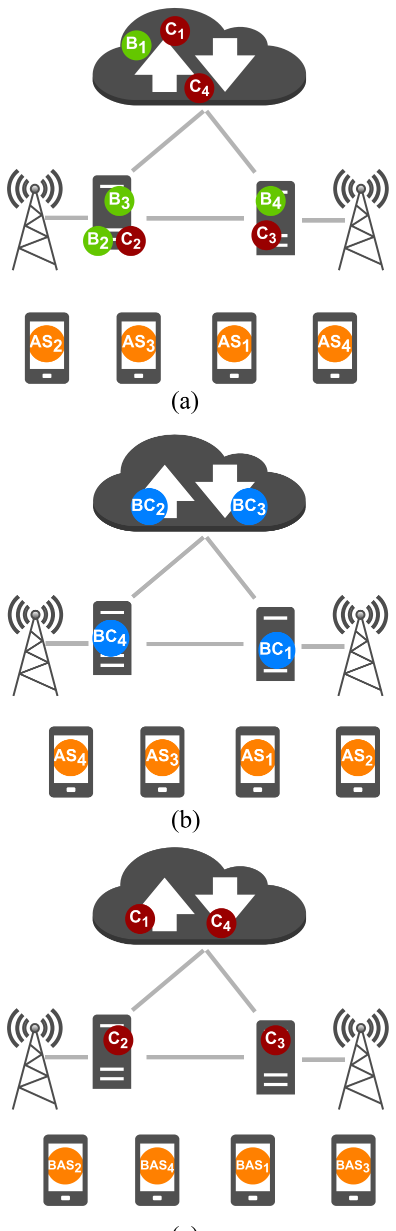

Fig. 5. Visual representation of the pulverized deployments under test. In $B \downdownarrows C$, behavior and communication components are forced to be hosted on the same host, in $B \rightleftarrows C$ they can be located on different hosts, and in C only the communication component is hosted on either edge or cloud.

3) *$B \downdownarrows C$ With $P_{☁} = 0$:* An application designed from the beginning to delegate to the edge servers everything but sensing and actuation (end devices are considered *thin*) and communicate through the edge.
4) *$B \downdownarrows C$ With $P_{☁} = 1$:* An application designed from the beginning to delegate to the cloud everything but sensing and actuation (end devices are considered *thin*) and communicate through the cloud.

End devices move following a *nomadic migration model*: they spend some time in the proximity of an edge server, then they can migrate elsewhere. The time spent at each destination depends on its *attractiveness* parameter (specified in EdgeCloudSim configuration files), while all destinations have the same probability of being reached.

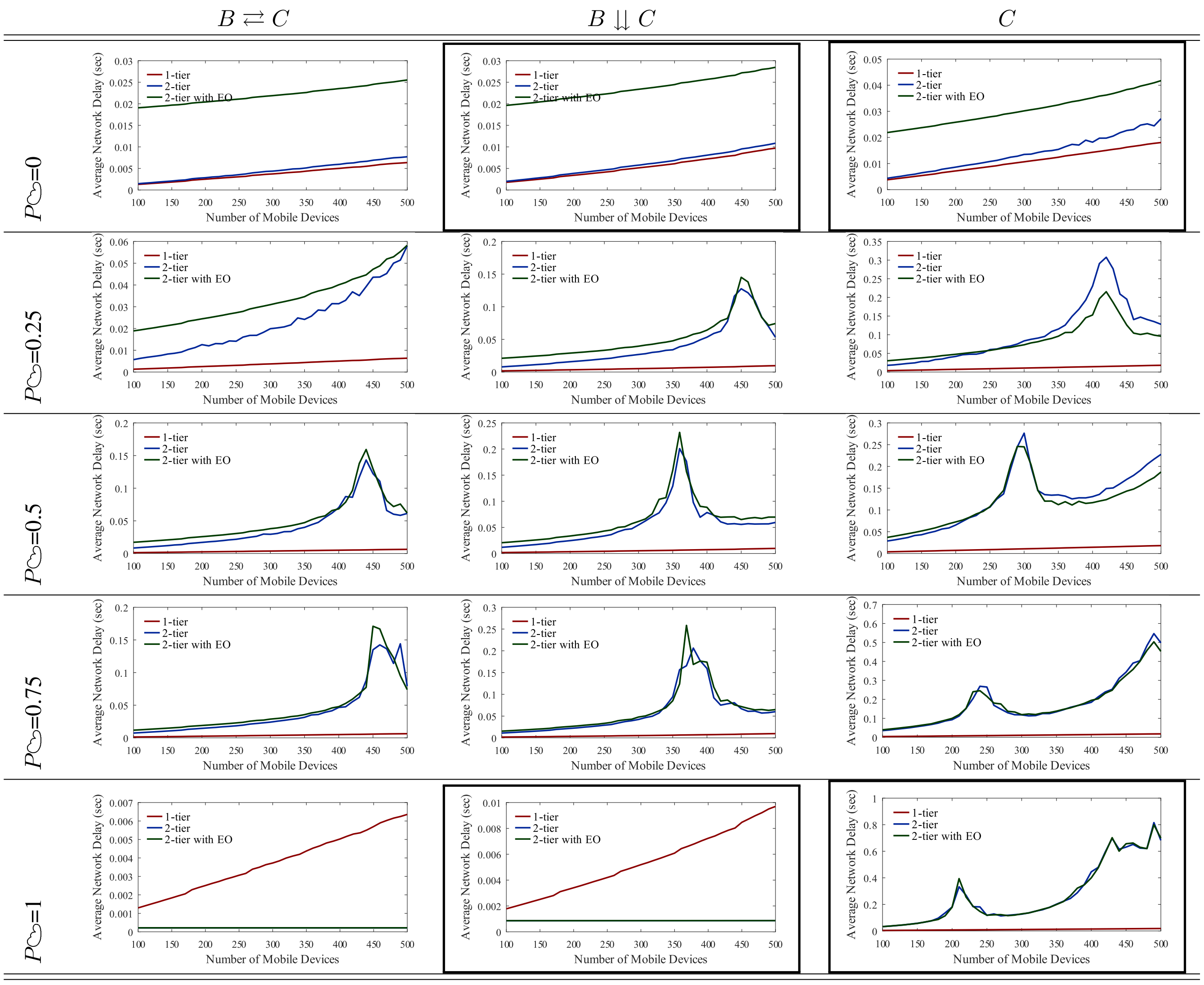

Fig. 6. Average network delay by deployment type (columns) and P☁ (rows). The last two columns ($B \downdownarrows C$and C) of the first (P☁ $= 0$) and the last (P☁ $= 1$) rows, surrounded by black boxes, can be considered baselines, as these data could be generated and studied with classic approaches too.

The whole experiment has been documented, automated, and published as opensource in a public repository[8] [52] to facilitate accessibility and reproduction. Unfortunately, EdgeCloudSim does not allow for seeding simulations: results obtained by re-executing the process will produce slightly different results.

B. Results

Simulation results are summarized in Fig. 6 for network delays and in Fig. 7 for the task failure rate. Data are the mean over seven simulation runs.

First, the proposed methodology *widens the design space* of the distributed application at hand: with traditional development methodologies, the application should have been designed from the start to work on either *thick* devices (C deployments), or with *thin* end devices, but with behavior and communication logics co-hosted ($B \downdownarrows C$ deployments); since single application parts get designed with the communication machinery in mind, either the cloud (P☁ $= 1$) or the edge (P☁ $= 0$) is used. The proposed methodology opens the door to many further possible deployment schemes, as all choices on how single components should communicate and where they can be deployed are *delayed* until after the system business logic design is complete.

Analysis of the data for the several deployment schemes show no clear dominance. Figures depicting the average network delay show a peak due to an increased rate of network failures: these data are to be considered along with the probability that tasks complete successfully. We observe, in fact, that in 2-tiered scenarios, most failures are cloud side, and, by comparing failures with the corresponding relative network delays, we see that decreasing average network delays match the growth of the failure rate. In 1-tiered scenarios (edge only), we observe a minor average network delay compared to the other scenarios—as expected—but a higher percentage of failed tasks, most likely due to saturation of edge servers' computational resources. For the specific scenario (multimedia

[8]https://github.com/aPlacuzzi/Experiment-2021-Pulverization-EdgeCloudSim

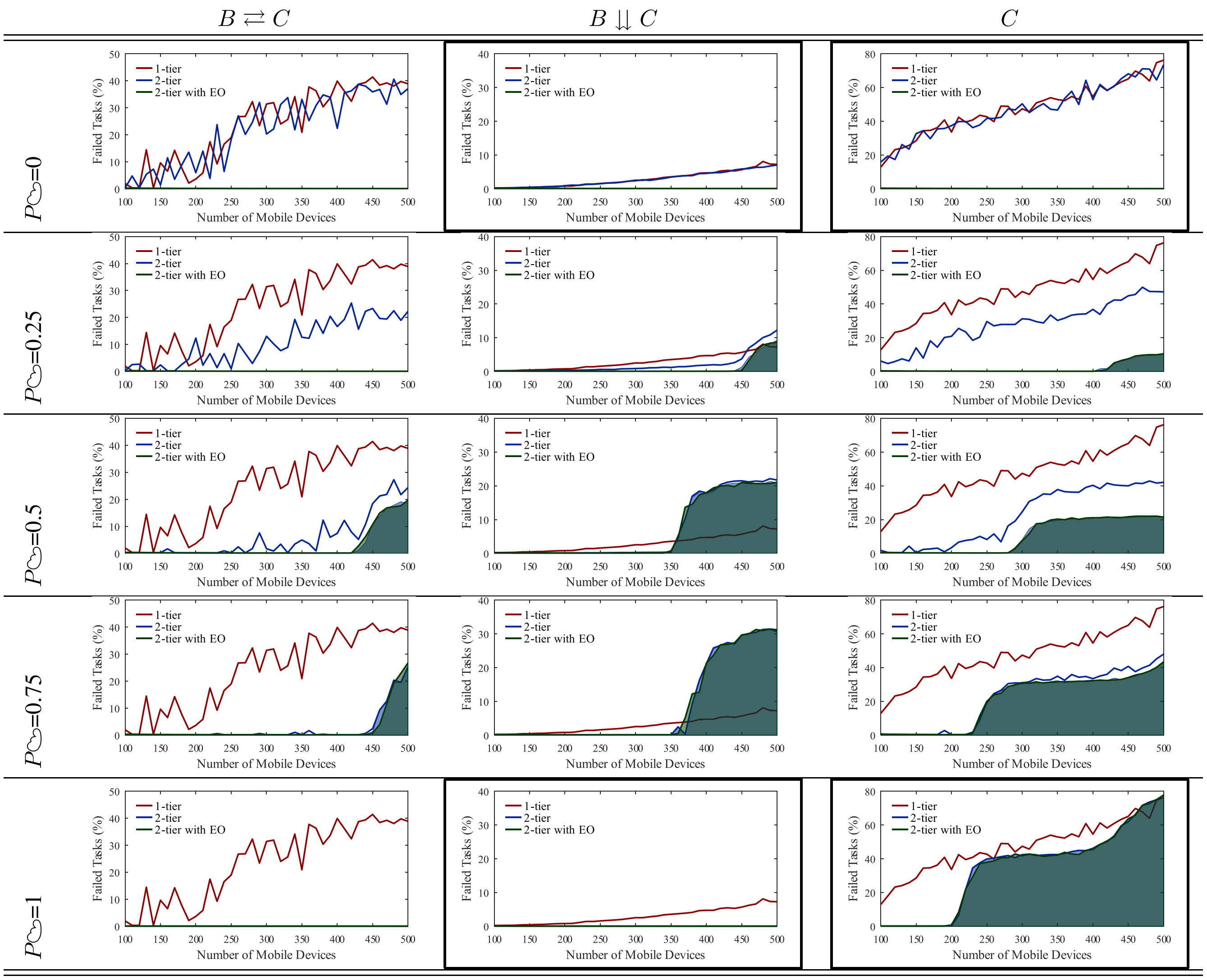

Fig. 7. Percent of failed tasks by deployment type (columns) and P☁ (rows). The last two columns ($B \downdownarrows C$and C) of the first (P☁ $= 0$) and the last (P☁ $= 1$) rows, surrounded by black boxes, can be considered baselines, as these data could be generated and studied with classic approaches too. Solid-filled areas represent the quota of failed tasks that were assigned to the cloud.

streaming), unless the streaming is intended for real-time reuse (e.g., augmented reality gaming), latency is likely more tolerable than failure. Data show that for a relatively low count of users (below 200) a 1-tiered architecture is a viable solution, provided that a $B \downdownarrows C$ deployment is performed, as $B \rightleftarrows C$ has a more erratic behavior due to the components being migrated separately, and C deployments stress the network too much. To support larger systems, a 2-tiered architecture is necessary, and in this case a $B \rightleftarrows C$ deployment with a low P☁ seems to be the most scalable solution. Adding an edge orchestrator in this situation is not particularly helpful, as it provides better performance only when the system is not under stress, while in heavy load configurations, the performance of an architecture with or without orchestrator tends to converge.

V. Conclusion

The development of collective, dynamic, heterogeneous and scalable IoT systems in complex and uncertain scenarios is an engineering task as important as challenging: from the 1) functional viewpoint, they require programming paradigms that inherently support essential features of autonomy, decentralization and adaptiveness and 2) deployment viewpoint, due to the entanglement of different factors related to computation, networking, and mobility aspects, they can be mapped in several, alternative, and suboptimal settings, with the risk of ineffective deployments and reconfigurations. Therefore, computing paradigms for expressing edge intelligence by separating logic and deployment planes as well as simulation tools for preliminary and comprehensively evaluating different target deployment configurations are key enablers to effectively and efficiently develop complex IoT systems.

Along this research direction, in this article, we have presented a methodology and a toolchain for capturing candidate deployments of smart edge services and predicting, by simulation, their performance and cost. In particular, we tailored this approach to aggregate computing, since its architecture enables pulverization (namely, partitioning into logical components and deployment units), spread on IoT-Edge-Fog-Cloud infrastructural continuum, and accordingly simulated on EdgeCloudSim. We have then demonstrated the potential of our approach to an edge multimedia streaming case study which mirrors the challenges and requirements of complex IoT systems (large scale, dynamicity, and adaptivity) and demands edge intelligence for load-balancing operations through dynamic clustering. In particular, we showed that, starting from the same pulverized specification, depending on the target deployment of our components, we can obtain systems with different performances. Each type of deployment works better under some circumstances depending, for example, on the expected number of users and the availability of computational and network resources. Since our methodology and the related prototype tool enables an *a priori* estimation of the deployed system performance, they can be exploited by operation teams to select the most suitable target platform for the system deployment or can be integrated into an automated deployment pipeline as an additional quality control step.

In the future, we plan to realize a variant of the presented methodology where the simulation step is substituted with an optimal method, similar to the ones proposed in [45], [47], and [49]. Moreover, we intend to perform further evaluations about both the methodology and other use cases.

References

[1] Z. Zhou, X. Chen, E. Li, L. Zeng, K. Luo, and J. Zhang, "Edge intelligence: Paving the last mile of artificial intelligence with edge computing," *Proc. IEEE*, vol. 107, no. 8, pp. 1738–1762, Aug. 2019. [Online]. Available: https://doi.org/10.1109/JPROC.2019.2918951

[2] S. Deng, H. Zhao, W. Fang, J. Yin, S. Dustdar, and A. Y. Zomaya, "Edge intelligence: The confluence of edge computing and artificial intelligence," *IEEE Internet Things J.*, vol. 7, no. 8, pp. 7457–7469, Aug. 2020. [Online]. Available: https://doi.org/10.1109/JIOT.2020.2984887

[3] R. Casadei, M. Viroli, G. Audrito, D. Pianini, and F. Damiani, "Engineering collective intelligence at the edge with aggregate processes," *Eng. Appl. Artif. Intell.*, vol. 97, Jan. 2021, Art. no. 104081. [Online]. Available: https://doi.org/10.1016/j.engappai.2020.104081

[4] V. Karagiannis and S. Schulte, "Distributed algorithms based on proximity for self-organizing fog computing systems," *Pervasive Mobile Comput.*, vol. 71, Feb. 2021, Art. no. 101316. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1574119220301437

[5] R. Casadei and M. Viroli, "Coordinating computation at the edge: A decentralized, self-organizing, spatial approach," in *Proc. 4th Int. Conf. Fog Mobile Edge Comput. FMEC*, Rome, Italy, Jun. 2019, pp. 60–67. [Online]. Available: https://doi.org/10.1109/FMEC.2019.8795355

[6] D. Pianini, R. Casadei, M. Viroli, and A. Natali, "Partitioned integration and coordination via the self-organising coordination regions pattern," *Future Gener. Comput. Syst.*, vol. 114, pp. 44–68, Jan. 2021. [Online]. Available: https://doi.org/10.1016/j.future.2020.07.032

[7] R. Casadei, G. Fortino, D. Pianini, W. Russo, C. Savaglio, and M. Viroli, "Modelling and simulation of opportunistic IoT services with aggregate computing," *Future Gener. Comput. Syst.*, vol. 91, pp. 252–262, Feb. 2019. [Online]. Available: https://doi.org/10.1016/j.future.2018.09.005

[8] R. Casadei, G. Fortino, D. Pianini, W. Russo, C. Savaglio, and M. Viroli, "A development approach for collective opportunistic edge-of-things services," *Inf. Sci.*, vol. 498, pp. 154–169, Sep. 2019. [Online]. Available: https://doi.org/10.1016/j.ins.2019.05.058

[9] R. Casadei, C. Tsigkanos, M. Viroli, and S. Dustdar, "Engineering resilient collaborative edge-enabled IoT," in *Proc. IEEE Int. Conf. Services Comput. (SCC)*, Milan, Italy, Jul. 2019, pp. 36–45. [Online]. Available: https://doi.org/10.1109/SCC.2019.00019

[10] R. Casadei, D. Pianini, A. Placuzzi, M. Viroli, and D. Weyns, "Pulverization in cyber-physical systems: Engineering the self-organizing logic separated from deployment," *Future Internet*, vol. 12, no. 11, p. 203, 2020. [Online]. Available: https://doi.org/10.3390/fi12110203

[11] G. Fortino, C. Savaglio, G. Spezzano, and M. Zhou, "Internet of Things as system of systems: A review of methodologies, frameworks, platforms, and tools," *IEEE Trans. Syst., Man, Cybern., Syst.*, vol. 51, no. 1, pp. 223–236, Jan. 2021. [Online]. Available: https://doi.org/10.1109/TSMC.2020.3042898

[12] M. Artac, T. Borovsak, E. D. Nitto, M. Guerriero, and D. A. Tamburri, "DevOps: Introducing infrastructure-as-code," in *Proc. 39th Int. Conf. Softw. Eng. (ICSE)*, Buenos Aires, Argentina, May 2017, pp. 497–498. [Online]. Available: https://doi.org/10.1109/ICSE-C.2017.162

[13] D. Pianini, M. Viroli, and J. Beal, "Protelis: Practical aggregate programming," in *Proc. 30th Annu. ACM Symp. Appl. Comput.*, Salamanca, Spain, Apr. 2015, pp. 1846–1853. [Online]. Available: https://doi.org/10.1145/2695664.2695913

[14] C. Sonmez, A. Ozgovde, and C. Ersoy, "EdgeCloudSim: An environment for performance evaluation of edge computing systems," *Trans. Emerg. Telecommun. Technol.*, vol. 29, no. 11, 2018, Art. no. e3493. [Online]. Available: https://doi.org/10.1002/ett.3493

[15] J. Beal, D. Pianini, and M. Viroli, "Aggregate programming for the Internet of Things," *Computer*, vol. 48, no. 9, pp. 22–30, 2015. [Online]. Available: https://doi.org/10.1109/MC.2015.261

[16] A. Bucchiarone and M. Mongiello, "Ten years of self-adaptive systems: From dynamic ensembles to collective adaptive systems," in *From Software Engineering to Formal Methods and Tools, and Back* (LNCS 11865), M. H. ter Beek, A. Fantechi, and L. Semini, Eds. Cham, Switzerland: Springer, 2019, pp. 19–39. [Online]. Available: https://doi.org/10.1007/978-3-030-30985-5_3

[17] A. Aldini, "Design and verification of trusted collective adaptive systems," *ACM Trans. Model. Comput. Simulat.*, vol. 28, no. 2, pp. 1–9, 2018. [Online]. Available: https://doi.org/10.1145/3155337

[18] D. B. Abeywickrama, N. Bicocchi, M. Mamei, and F. Zambonelli, "The SOTA approach to engineering collective adaptive systems," *Int. J. Softw. Tools Technol. Transf.*, vol. 22, no. 4, pp. 399–415, 2020. [Online]. Available: https://doi.org/10.1007/s10009-020-00554-3

[19] J. Beal, K. Usbeck, J. P. Loyall, M. Rowe, and J. M. Metzler, "Adaptive opportunistic airborne sensor sharing," *ACM Trans. Auton. Adapt. Syst.*, vol. 13, no. 1, pp. 1–6, 2018. [Online]. Available: https://doi.org/10.1145/3179994

[20] S. S. Clark, J. Beal, and P. P. Pal, "Distributed recovery for Enterprise services," in *Proc. IEEE 9th Int. Conf. Self Adapt. Self Org. Syst.*, Cambridge, MA, USA, Sep. 2015, pp. 111–120. [Online]. Available: https://doi.org/10.1109/SASO.2015.19

[21] R. D. Nicola, S. Jähnichen, and M. Wirsing, "Rigorous engineering of collective adaptive systems: Special section," *Int. J. Softw. Tools Technol. Transf.*, vol. 22, no. 4, pp. 389–397, 2020. [Online]. Available: https://doi.org/10.1007/s10009-020-00565-0

[22] J. Beal, S. Dulman, K. Usbeck, M. Viroli, and N. Correll. "Organizing the aggregate: Languages for spatial computing." 2012. [Online]. Available: http://arxiv.org/abs/1202.5509

[23] M. Viroli, J. Beal, F. Damiani, G. Audrito, R. Casadei, and D. Pianini, "From distributed coordination to field calculus and aggregate computing," *J. Logic. Algebr. Methods Program.*, vol. 109, Dec. 2019, Art. no. 100486. [Online]. Available: https://doi.org/10.1016/j.jlamp.2019.100486

[24] R. Casadei, M. Viroli, G. Audrito, and F. Damiani, "FScaFi: A core calculus for collective adaptive systems programming," in *Leveraging Applications of Formal Methods, Verification and Validation: Engineering Principles*, (LNCS 12477), T. Margaria and B. Steffen, Eds. Cham, Switzerland: Springer, 2020, pp. 344–360. [Online]. Available: https://doi.org/10.1007/978-3-030-61470-6_21

[25] G. Audrito, R. Casadei, F. Damiani, and M. Viroli, "Compositional blocks for optimal self-healing gradients," in *Proc. 11th IEEE Int. Conf. Self Adapt. Self Org. Syst. (SASO)*, Tucson, AZ, USA, Sep. 2017, pp. 91–100. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/SASO.2017.18

[26] J. Liu, E. Ahmed, M. Shiraz, A. Gani, R. Buyya, and A. Qureshi, "Application partitioning algorithms in mobile cloud computing: Taxonomy, review and future directions," *J. Netw. Comput. Appl.*, vol. 48, pp. 99–117, Feb. 2015. [Online]. Available: https://doi.org/10.1016/j.jnca.2014.09.009

[27] M. J. North, N. T. Collier, and J. R. Vos, "Experiences creating three implementations of the repast agent modeling toolkit," *ACM Trans. Model. Comput. Simulat.*, vol. 16, no. 1, pp. 1–25, 2006. [Online]. Available: https://doi.org/10.1145/1122012.1122013

[28] E. Sklar, "NetLogo, a multi-agent simulation environment," *Artif. Life*, vol. 13, no. 3, pp. 303–311, 2007. [Online]. Available: https://doi.org/10.1162/artl.2007.13.3.303

[29] D. Pianini, S. Montagna, and M. Viroli, "Chemical-oriented simulation of computational systems with ALCHEMIST," *J. Simulat.*, vol. 7, no. 3, pp. 202–215, 2013. [Online]. Available: https://doi.org/10.1057/jos.2012.27

[30] M. Viroli, R. Casadei, and D. Pianini, "Simulating large-scale aggregate MASs with alchemist and scala," in *Proc. IEEE Feder. Conf. Comput. Sci. Inf. Syst. (FedCSIS)*, vol. 8, Sep. 2016, pp. 1495–1504. [Online]. Available: https://doi.org/10.15439/2016F407

[31] D. Pianini, A. Elzanaty, A. Giorgetti, and M. Chiani, "Emerging distributed programming paradigm for cyber-physical systems over LoRaWANs," in *Proc. IEEE Globecom Workshops GC Wkshps*, Abu Dhabi, UAE, Dec. 2018, pp. 1–6. [Online]. Available: https://doi.org/10.1109/GLOCOMW.2018.8644518

[32] A. Varga, "OMNeT++," in *Modeling and Tools for Network Simulation*. Berlin, Germany: Springer, 2010, pp. 35–59. [Online]. Available: https://doi.org/10.1007/978-3-642-12331-3_3

[33] G. F. Riley and T. R. Henderson, "The ns-3 network simulator," in *Modeling and Tools for Network Simulation*. Berlin, Germany: Springer, 2010, pp. 15–34. [Online]. Available: https://doi.org/10.1007/978-3-642-12331-3_2

[34] P. Levis, N. Lee, M. Welsh, and D. E. Culler, "TOSSIM: Accurate and scalable simulation of entire tinyOS applications," in *Proc. 1st Int. Conf. Embedded Netw. Sensor Syst. (SenSys)*, Los Angeles, CA, USA, Nov. 2003 pp. 126–137. [Online]. Available: https://doi.org/10.1145/958491.958506

[35] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. F. D. Rose, and R. Buyya, "CloudSim: A toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms," *Softw. Pract. Exp.*, vol. 41, no. 1, pp. 23–50, 2011. [Online]. Available: https://doi.org/10.1002/spe.995

[36] A. C. H. Chow and B. P. Zeigler, "Parallel DEVS: A parallel, hierarchical, modular, modeling formalism," in *Proc. 26th Conf. Winter Simulat. (WSC)*, Lake Buena Vista, FL, USA, Dec. 1994, pp. 716–722. [Online]. Available: https://doi.org/10.1109/WSC.1994.717419

[37] J. Byrne *et al.*, "RECAP simulator: Simulation of cloud/edge/fog computing scenarios," in *Proc. Winter Simulat. Conf. (WSC)*, Las Vegas, NV, USA, Dec. 2017, pp. 4568–4569. [Online]. Available: https://doi.org/10.1109/WSC.2017.8248208

[38] M. Ashouri, F. Lorig, P. Davidsson, and R. Spalazzese, "Edge computing simulators for IoT system design: An analysis of qualities and metrics," *Future Internet*, vol. 11, no. 11, p. 235, 2019. [Online]. Available: https://doi.org/10.3390/fi11110235

[39] S. Svorobej *et al.*, "Simulating fog and edge computing scenarios: An overview and research challenges," *Future Internet*, vol. 11, no. 3, p. 55, 2019. [Online]. Available: https://doi.org/10.3390/fi11030055

[40] "Systems and software engineering—Systems and software quality requirements and evaluation (SQuaRE)—Measurement of system and software product quality," Int. Org. Stand., Geneva, Switzerland, Rep. TR 25023, 2016. [Online]. Available: https://www.iso.org/standard/35747.html

[41] M. Ayoubi, M. Ramezanpour, and R. Khorsand, "An autonomous IoT service placement methodology in fog computing," *Softw. Pract. Exp.*, vol. 51, no. 5, pp. 1097–1120, Dec. 2020. [Online]. Available: https://doi.org/10.1002/spe.2939

[42] G. Tanganelli, L. Cassano, A. Miele, and C. Vallati, "A methodology for the design and deployment of distributed cyber-physical systems for smart environments," *Future Gener. Comput. Syst.*, vol. 109, pp. 420–430, Aug. 2020. [Online]. Available: https://doi.org/10.1016/j.future.2020.02.047

[43] S. Venticinque and A. Amato, "A methodology for deployment of IoT application in fog," *J. Ambient Intell. Humanized Comput.*, vol. 10, no. 5, pp. 1955–1976, Apr. 2018. [Online]. Available: https://doi.org/10.1007/s12652-018-0785-4

[44] C. Avasalcai, B. Zarrin, and S. Dustdar, "EdgeFlow—Developing and deploying latency-sensitive IoT edge applications," *IEEE Internet Things J.*, vol. 9, no. 5, pp. 3877–3888, Mar. 2022. [Online]. Available: https://doi.org/10.1109/jiot.2021.3101449

[45] R. Deng, R. Lu, C. Lai, T. H. Luan, and H. Liang, "Optimal workload allocation in fog-cloud computing towards balanced delay and power consumption," *IEEE Internet Things J.*, vol. 3, no. 6, pp. 1171–1181, Dec. 2016. [Online]. Available: https://doi.org/10.1109/jiot.2016.2565516

[46] A. da Silva Veith, M. D. de Assuncao, and L. Lefevre, "Latency-aware strategies for deploying data stream processing applications on large cloud-edge infrastructure," *IEEE Trans. Cloud Comput.*, early access, Jul. 20, 2021. [Online]. Available: https://doi.org/10.1109/tcc.2021.3097879

[47] B. Cao, Q. Wei, Z. Lv, J. Zhao, and A. K. Singh, "Many-objective deployment optimization of edge devices for 5G networks," *IEEE Trans. Netw. Sci. Eng.*, vol. 7, no. 4, pp. 2117–2125, Oct.–Dec. 2020. [Online]. Available: https://doi.org/10.1109/tnse.2020.3008381

[48] Y. Dong, G. Xu, Y. Ding, X. Meng, and J. Zhao, "A 'joint-me' task deployment strategy for load balancing in edge computing," *IEEE Access*, vol. 7, pp. 99658–99669, 2019. [Online]. Available: https://doi.org/10.1109/access.2019.2928582

[49] A. Brogi, S. Forti, and A. Ibrahim, *Predictive Analysis to Support Fog Application Deployment*. Hoboken, NJ, USA: Wiley, Jan. 2019, pp. 191–221. [Online]. Available: https://doi.org/10.1002/9781119525080.ch9

[50] M. Ficco, C. Esposito, Y. Xiang, and F. Palmieri, "Pseudo-dynamic testing of realistic edge-fog cloud ecosystems," *IEEE Commun. Mag.*, vol. 55, no. 11, pp. 98–104, Nov. 2017. [Online]. Available: https://doi.org/10.1109/mcom.2017.1700328

[51] X. Chen *et al.*, "iDiSC: A new approach to IoT-data-intensive service components deployment in edge-cloud-hybrid system," *IEEE Access*, vol. 7, pp. 59172–59184, 2019. [Online]. Available: https://doi.org/10.1109/access.2019.2915020

[52] A. Placuzzi and D. Pianini. "aPlacuzzi/experiment-2021-Pulverization-EdgeCloudSim: Release 0.1.0-2021-04-29T163149." 2021. [Online]. Available: https://zenodo.org/record/4727996

[53] P. Louridas, "Static code analysis," *IEEE Softw.*, vol. 23, no. 4, pp. 58–61, Jul./Aug. 2006. [Online]. Available: https://doi.org/10.1109/MS.2006.114

[54] G. Audrito, M. Viroli, F. Damiani, D. Pianini, and J. Beal, "A higher-order calculus of computational fields," *ACM Trans. Comput. Logic*, vol. 20, no. 1, pp. 1–5, 2019. [Online]. Available: https://doi.org/10.1145/3285956

[55] M. Eysholdt and H. Behrens, "Xtext: Implement your language faster than the quick and dirty way," in *Proc. 25th Annu. ACM SIGPLAN Conf. Object Orient. Program. Syst. Lang. Appl. (SPLASH/OOPSLA)*, Oct. 2010, pp. 307–309. [Online]. Available: https://doi.org/10.1145/1869542.1869625

[56] B. K. Bershad, R. P. Draves, and A. Forin, "Using microbenchmarks to evaluate system performance," in *Proc. IEEE 3rd Workshop Workstation Oper. Syst.*, 1992, pp. 1–6. [Online]. Available: https://doi.org/10.1109/wwos.1992.275671

[57] K. Hoste, A. Georges, and L. Eeckhout, "Automated just-in-time compiler tuning," in *Proc. 8th Int. Symp. Code Gener. Optim. (CGO)*, Toronto, ON, Canada, Apr. 2010, pp. 62–72. [Online]. Available: https://doi.org/10.1145/1772954.1772965

[58] B. Campbell, "Terraform in-depth," in *The Definitive Guide to AWS Infrastructure Automation*. London, U.K.: Apress, Dec. 2019, pp. 123–203. [Online]. Available: https://doi.org/10.1007/978-1-4842-5398-4_4

[59] D. Pianini. "DanySK/experiment-2019-FGCS-self-integration: 1.0.0." 2021. [Online]. Available: https://zenodo.org/record/4568197

[60] S. Narayan, C. Jayawardena, J. Wang, W. Ma, and G. Geetu, "Performance test of IEEE 802.11ac wireless devices," in *Proc. IEEE Int. Conf. Comput. Commun. Inf. (ICCCI)*, Jan. 2015, pp. 1–6. [Online]. Available: https://doi.org/10.1109/iccci.2015.7218076

[61] D. Newell, P. Davies, R. Wade, P. Decaux, and M. Shama, "Comparison of theoretical and practical performances with 802.11n and 802.11ac wireless networking," in *Proc. 31st Int. Conf. Adv. Inf. Netw. Appl. Workshops (WAINA)*, Mar. 2017, pp. 710–715. [Online]. Available: https://doi.org/10.1109/waina.2017.113

Roberto Casadei (Member, IEEE) received the Ph.D. degree in computer science and engineering from Alma Mater Studiorum–Università di Bologna, Cesena, Italy, in 2020.

He is a Postdoctoral Researcher and an Adjunct Professor with Alma Mater Studiorum–Università di Bologna. He has over 35 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 opensource ScaFi aggregate programming toolkit. His research interests revolve around software engineering and distributed artificial intelligence.

Dr. Casadei has served as a Workshop Chair for the eCAS Workshop, as an Organizing or PC Member of multiple conferences, including COORDINATION and ACSOS, and as a reviewer for renowned international journals.

Giancarlo Fortino (Fellow, IEEE) received the Ph.D. degree in computer and systems engineering from the University of Calabria, Rende, Italy, in 2000.

He is a Full Professor of Computer Engineering with the Department of Informatics, Modeling, Electronics, and Systems, University of Calabria (UniCal). He has authored 550+ papers in int'l journals, conferences and books. His research interests include wearable computing systems, Internet of Things, and cyber-security.

Prof. Fortino is Highly Cited Researcher 2002–2021 in Computer Science. He is the (Founding) Series Editor of the IEEE Press Book Series on Human–Machine Systems and of the Springer Internet of Things series and is AE of premier IEEE Transactions. He is a cofounder and the CEO of SenSysCal S.r.l., a Unical spinoff focused on innovative IoT systems. He is currently a member of the IEEE SMCS BoG and the Chair of the IEEE SMCS Italian Chapter.

Danilo Pianini (Member, IEEE) received the Ph.D. degree in computer science and engineering from the University of Bologna, Cesena, Italy, in 2015.

He is a Postdoctoral Researcher with the Department of Computer Science and Engineering, University of Bologna. He is the Lead Designer of dozens of opensource software tools, including the Alchemist simulation platform and the Protelis aggregate programming language. His main research interests include simulation, (self-organizing) coordination, aggregate computing, pervasive systems, software engineering, agile techniques, and DevOps. He has published over 50 articles in international journals and conferences on those subjects.

Andrea Placuzzi received the master's degree in computer science and engineering from the University of Bologna, Cesena, Italy, in 2020.

His research topics include software engineering, Internet of Things, self-organization, and simulation.

Claudio Savaglio (Member, IEEE) received the Ph.D. degree in ICT from the University of Calabria, Rende, Italy, in 2018.

He is a Researcher with ICAR-CNR Institute, Arcavacata, Italy. He was a Visiting Researcher with the University of Texas at Dallas, Richardson, TX, USA, in 2013; New Jersey Institute of Technology, Newark, NJ, USA, in 2016; and Universitat Politècnica de València, Valencia, Spain, in 2017. He has authored over 50 publications in international journals, conferences, and books. His research interests include Internet of Things, network simulation, edge intelligence, and agent-oriented development methodologies.

Mirko Viroli (Senior Member, IEEE) received the Ph.D. degree in electronics and computer engineering from the University of Bologna, Cesena, Italy, in 2003.

He is a Full Professor of Computer Engineering with the University of Bologna. He is an expert in foundations of computer science and programming, object-oriented programming, advanced software development, software engineering, and self-adaptive/self-organizing pervasive computing systems. His Google Scholar h-index is 48 with more than 8000 citations. He has authored more than 300 papers, of which more than 70 are on international journals.

Prof. Viroli is a member of the editorial board of *IEEE Software Magazine* and was a Program Chair of the ACM Symposium on Applied Computing (SAC 2008 and 2009) and IEEE Self-Adaptive and Self-Organizing Systems (SASO 2014) conferences.

Open Access funding provided by 'Università di Bologna' within the CRUI-CARE Agreement