Declarative Application Management in the Fog

A Bacteria-Inspired Decentralised Approach

Antonio Brogi [ORCID] · **Stefano Forti** [ORCID] ·
Carlos Guerrero [ORCID] · **Isaac Lera** [ORCID]

Abstract Orchestrating next-gen applications over heterogeneous resources along the Cloud-IoT continuum calls for new strategies and tools to enable scalable and application-specific managements. Inspired by the self-organisation capabilities of bacteria colonies, we propose a declarative, fully decentralised application management solution, targeting pervasive opportunistic Cloud-IoT infrastructures. We present a customisable declarative implementation of the approach and validate its scalability through simulation over motivating scenarios, also considering end-user's mobility and the possibility to enforce application-specific management policies for different (classes of) applications.

Author names are ordered alphabetically. All authors have contributed equally to this study.

A. Brogi · S. Forti (✉)
Department of Computer Science,
University of Pisa, Pisa, Italy
e-mail: brogi@di.unipi.it

S. Forti
e-mail: stefano.forti@di.unipi.it

C. Guerrero · I. Lera
Department of Mathematics and Computer Science,
University of Balearic Islands, Palma, Spain
e-mail: carlos.guerrero@uib.es

I. Lera
e-mail: isaac.lera@uib.es

1 Introduction

In recent years, the problem of how to effectively manage distributed applications along the Cloud-IoT continuum [1–4] – i.e. in Fog computing settings – has gained increasing attention from the scientific community [5–7]. Addressing such a problem is challenging due to the scale and heterogeneity that characterise Fog infrastructures, and due to the plethora requirements that next-gen IoT applications need to meet at run-time (e.g. end-to-end latency, success response rate, load-balancing) [8]. In this context, new abstractions, methodologies and tools are needed to achieve two fundamental goals in Fog application management:

(i) *scalability* — to master the provably exp-time complexity [9] related to making informed decisions throughout the application lifecycle in large-scale settings, from first placement to undeployment, through scaling and migrations,

(ii) *flexibility* — to enable application operators to specify *ad-hoc* management policies capable of capturing a range of diverse application needs and to dynamically enforce such policies at run-time [6].

Most of the existing literature in the field of Fog application management has proposed centralised *one-size-fits-all* solutions to make informed decisions, based on a global view of the available infrastructure [5]. These solutions shine when dedicated computational resources are available, when optimisation metrics are fixed once for all, or when applications can be statically placed for long-enough periods — thus amortising the costs of decision-making (e.g. in smart-home, gaming, virtual reality applications) [10]. On the contrary, such solutions show their shortcomings when the available infrastructure is *opportunistic*, built by exploiting a large number of heterogeneous (mobile) resource-constrained devices, or when distinct (classes of) applications running over the same infrastructure require optimising different metrics. Examples of these include disaster recovery or mass-event applications (e.g. concerts), which prevent employing centralised decision-making, demanding instead *self-organisation capabilities* of application management [11–14].

In our previous work [15], we partly addressed the aforementioned limitations of centralised approaches by proposing a *fully decentralised* and *declarative* solution for M*anaging* A*pplications* R*unning* I*n* O*pportunistic* Fog scenarios, and its open-source Prolog prototype MARIO. Being fully decentralised, MARIO is *scalable*. Notably, it is also highly *flexible*, since it enables application operators to declaratively specify *ad hoc* application management policies, using a set of simple operations that trigger based on available monitored data. MARIO assumes application management agents to be in charge of single application instances. Besides, it assumes they can access monitored data on the nodes in their direct vicinity (viz. free hardware at neighbouring nodes), and on all nodes traversed by user requests targeting their application instance (viz. request rate, free hardware in traversed nodes, end-to-end path latencies).

In this article, pursuing the effort towards fully decentralised declarative application management in the Fog, we extend the preliminary modelling and results of [15], and we open-source a new prototype[1],

MARIO$_2$. The most important novelties featured by MARIO$_2$ are:

Biological Parallel and Formal Operational Semantics –
By establishing a parallel between our solution and the biological behaviour of colonies of bacteria, we refine the set of management operations that can be used to declare management policies, and we propose a formal operational semantics of such operations to formally characterise the emerging behaviour of MARIO$_2$ application management agents running over a Fog infrastructure.

Limited Contextually Available Data – We reduce the amount of information available at application management agents, by limiting it to the free hardware at their host node and to the request rates from neighbouring nodes along with the end-to-end latency they experience. This dramatically reduces the information that node managers disclose to application management agents, placing MARIO$_2$ into more realistic Fog settings compared to MARIO.

New Policies and Experiments with User Mobility –
We propose four new management policies based on the new modelling and we extensively simulate them in YAFS [16] over a large-scale lifelike scenario, based on real data about taxi traces in Rome, Italy [17]. The obtained results confirm the scalability and flexibility of the new MARIO$_2$, also proving it successful in managing applications in presence of end-users' mobility.

Last, but not least, this article presents an extended discussion of related work to capture recent advances in the field of decentralised management solutions in Fog and Cloud-IoT settings.

The rest of this article is organised as follows. Section 2 describes the parallel among the emergent behaviour of bacteria colonies and our declarative fully-decentralised management solution, providing a formal operational semantics of the behaviour of MARIO$_2$ application management agents. Section 3 describes a Prolog prototype implementation of such a solution and four declarative (workload- and latency-aware) application management policies it supports. Those policies are showcased and thoroughly assessed via simulations over a lifelike scenario with user mobility in Section 4. Finally, Section 5 discusses

---

[1]Available at https://github.com/acsicuib/MARIO/tree/MarioII.

some related work and Section 6 draws some concluding remarks and outlines some possible directions for future work.

2 Modelling Bacteria-inspired Declarative Application Management

2.1 Biological Interpretation of MARIO$_2$

Seeking to bring the self-organisation capabilities of biological systems in opportunistic Fog application management, MARIO$_2$ strongly inspires from the behaviour of bacteria. Indeed, MARIO$_2$ relies upon fully-decentralised agents that perform three simple customisable operations, mimicking primitive biological functionalities. In this section, we illustrate the correspondence between those simple operations and the emergent behaviour of bacteria, and we provide a formal operational semantics of the MARIO$_2$ decentralised management solution.

Application management agents, as bacteria, live in a *host environment* represented by a Cloud-IoT infrastructure. Infrastructure nodes correspond to *places* in the host environment where contextual deployment conditions (e.g. available hardware resources, client requests to the application, end-to-end application latencies, resource management policies) can be more or less favourable for the considered application, and consequently trigger different management operations. Figure 1 sketches the parallel between the considered biological functionalities of bacteria [18] and the management operations featured by MARIO$_2$ agents. Particularly:

Motility   Bacteria can move around by means of *flagella*, i.e. lashes protruding from their bodies. Similarly, application instances in the MARIO$_2$ solution can perform *migrations* from one deployment node to another according to their management policy and to the host environment around them. Figure 1(a) shows the analogy between bacteria mobility and application migration in MARIO$_2$.

Binary Fission   When their environment conditions are favourable, bacteria asexually reproduce via *binary fission*, i.e. parent cells divide into two identical daughter cells. Similarly, application instances
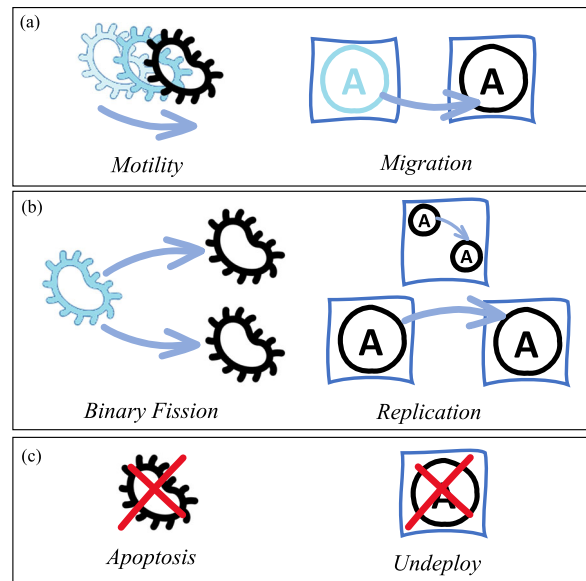

Fig. 1 Bacteria functionalities and corresponding MARIO$_2$ operations

in the MARIO$_2$ solution can replicate into a new identical application instance, on either the same node or on a node in their vicinity,[2] depending on which one represents the more favourable place in their host environment according to the enforced application management policy. Figure 1(b) shows the analogy between bacteria binary fission and application replication in MARIO$_2$, when an application instance replicates in the same or in a different target node.

Apoptosis   Bacteria feature a form of programmed cell death called *apoptosis*, i.e. bacteria can suicide when death of an individual translates into benefits for its colony [19]. Similarly, application instances in MARIO$_2$ can decide to autonomically *undeploy* from their host node, when needed according to their management policy. Figure 1(c) shows the analogy between bacteria apoptosis and application instance undeployment in MARIO$_2$.

---

[2]Binary fission onto a neighbouring node enables application instances to replicate – e.g. to handle increased workloads – even when their current node lacks the needed resources. Note that such a behaviour could not be achieved by composing a replication onto the current node with a subsequent migration.

Last, but not least, bacteria are capable of *sensing* their environment to properly react with some of the illustrated biological functionalities, when needed. Similarly, we assume that nodes feature decentralised *monitoring* capabilities (e.g. those of FogMon [20, 21] or of the monitoring tools surveyed in [22]), enabling application management agents to access information about their current deployment context. Pursuing our analogy, bacteria sensing capabilities correspond to the fact that application management agents can access information on the hardware resources at their current deployment node and on the amount of client requests they receive from each neighbouring node, along with the client-to-application latency such requests experience.

2.2 Emerging Behaviour of MARIO$_2$ Bacteria

We formally define here the emergent behaviour of MARIO$_2$ by means of a labelled transition system describing the evolution of application deployments over Cloud-IoT infrastructures.

Each application instance is associated with an *application management agent* (i.e. a *bacterion*), denoted by a tuple of the form $\langle i, a, p \rangle$ where

- i is the (unique) identifier of the application management agent,
- a is the managed application instance, and
- p is the management policy associated to the application instance, selecting one management action to be executed, based on contextually available data.

Infrastructure nodes are denoted by tuples of the form $\langle n, A_n, K_n, P_n \rangle$ where

- n is a (unique) node identifier,
- A_n is the set of application management agents running in node n,

- K_n is the contextual data available at node n to the application management agents in A_n, and
- P_n is (a predicate defining) the acceptance policy employed by node n to accept/reject deployment requests of application management agents.

An application management agent b running on a node n can decide to take different actions according to its management policy p applied to the contextually available data K_n. Namely an application management agent b can request: (1) to *migrate* (together with its associated application instance) to a neighbour node m of n, (2) to *replicate* itself in the same node n where it is running, or in a neighbour node m of n, by starting a new instance b' of b, or (3) to *undeploy* itself.

Rule (r1) describes the controlled evolution of a set of infrastructure nodes. The set **R** denotes the requests issued by the bacteria present in the nodes of the infrastructure as well as by human/autonomic application operators wishing to *deploy* new application management agents.

Each request of an agent b running in a node n is assessed by (the node manager of) n according to its acceptance management policy P_n. Requests to migrate or to replicate in a neighbour node m of n are also assessed by (the node manager of) m. Accepted requests **A** are all enacted in parallel, while non-accepted requests are inhibited and do not produce changes in the system configuration.

The set of accepted and inhibited requests labels the transition modelling the evolution of a set N of infrastructure nodes. The effect of the transition is that the set of infrastructure nodes N evolves into N' by updating the set of bacteria running in each node $n \in N$ according to the *deploy*, *migrate*, *replicate* and *undeploy* requests that were accepted.

$$
\begin{aligned}
\mathbf{R} = \{&\langle b, n, op \rangle \mid \langle n, A_n, K_n, P_n \rangle \in N \ \wedge \ b = \langle i, a, p \rangle \in A_n \ \wedge \ op = p(K_n)\} \\
&\cup \{\langle b, n, deploy \rangle \mid \text{operator requests to deploy new agent } b \text{ in node } n\}
\end{aligned}
$$

$$\mathbf{A} = \{x \mid x \in \mathbf{R} \ \wedge \ \forall n \in N : P_n(x)\} \subseteq \mathbf{R}$$

$$\mathbf{L} = \mathbf{A} \cup \{inhibited(w) \mid w \in \mathbf{R} - \mathbf{A}\}$$

$$(r1) \ \frac{}{N \xrightarrow{\mathbf{L}} N'}$$

where

$$N' = \{\langle n, A'_n, K_n, P_n \rangle \mid$$
$$\langle n, A_n, K_n, P_n \rangle \in N \wedge$$
$$A'_n = A_n \cup \{b \mid \langle b, n, deploy \rangle \in \mathbf{A}\}$$
$$\cup \{b \mid \langle b, \_, migrate(n) \rangle \in \mathbf{A}\}$$
$$\cup \{b' \mid \langle b, \_, replicate(n) \rangle \in \mathbf{A} \wedge b = \langle i, a, p \rangle \wedge b' = \langle j, a, p \rangle\}$$
$$- \{b \mid \langle b, n, undeploy \rangle \in \mathbf{A}\}$$
$$- \{b \mid \langle b, n, migrate(m) \rangle \in \mathbf{A}\}$$
$$\}$$

Finally, the set of contextually available data at each node can (arbitrarily) change at any time.

$$(r2) \quad \frac{\text{new data D available} \wedge N' = \{\langle n, A_n, K'_n, P_n \rangle \mid \langle n, A_n, K_n, P_n \rangle \in N \wedge K'_n = D_{\mid n}\}}{N \xrightarrow{\tau} N'}$$

where $D_{\mid n}$ denotes the set of data that become available to node n.

3 Prototyping Bacteria-inspired Declarative Application Management

In this section, after a short introduction to Prolog (Section 3.1), we discuss how the bio-inspired fully-decentralised declarative management solution described in Section 2 has been implemented in the MARIO$_2$ Prolog prototype (Section 3.2). Then, we describe four lifelike, increasingly complex, management policies that can be declared and enforced in MARIO$_2$ as declarative Prolog programs (Section 3.3).

3.1 Background: Prolog

Prolog is a logic programming language based on first-order logic.

A Prolog program is a finite set of *clauses* of the form:

$$\mathtt{A} : -B1, B2, ..., Bn. \tag{1}$$

where A is an atomic formula and where B1, B2, ..., Bn are (possibly negated) atomic formulas. Clauses with empty premise are also called *facts*.

The declarative reading of a clause like (1) is that A is true if B1, B2, ..., Bn are all true. The procedural reading of a clause like (1) is that if B1, B2, ..., Bn can be proven to hold (by SLD resolution), then A is proven to hold.

By convention, variables begin with upper-case letters, lists are denoted by square brackets, and negation by \+.

ISO Prolog implementations[3] allow to employ disjunctions (denoted by ;) in the premise of clauses, as for instance in:

```
A :- (B;C),D.
```

as a shorthand for:

$$\mathtt{A} : -B, D.$$
$$\mathtt{A} : -C, D.$$

ISO Prolog implementations also feature set predicates like

```
findall(Template, Goal, Result)
```

that can be exploited to find all successful solutions of `Goal` and collect the corresponding instantiations of `Template` in the list `Result`. If `Goal` has no solutions then `Result` is instantiated to the empty list.

3.2 Knowledge Representation in Prolog

3.2.1 Available Contextual Data

In this section, we describe the Prolog facts that application management agents can access to make informed decisions based on the requirements of the application that they are managing, on the status of their deployment node, and on the requests that their application instance is currently receiving.

Application management agents can access applications descriptors, detailing the requirements and capabilities of single-service applications denoted as in

```
service(ServiceId, RequiredHW,
        MaxRequestRate, MaxLatencyToClient).
```

where `ServiceId` is a unique application identifier, `RequiredHW` is the amount of hardware resources required by the application[4], `MaxRequestRate` is the maximum amount of client requests that a single application instance can handle, and `MaxLatencyToClient` is the maximum end-to-end latency that incoming requests can tolerate.

Application management agents can also access information on the capabilities of their current deployment node denoted as in

```
node(NodeId, AvailableHW).
```

where `NodeId` is a unique node identifier within the network (e.g. IP address) and `AvailableHW` is the free hardware that the node currently features.

Running application instances, managed by application management agents in MARIO$_2$, are denoted as in

```
serviceInstance(ServiceInstanceId, ServiceId, Node).
```

where `ServiceInstanceId` is a unique service instance identifier, `ServiceId` is the identifier of the application descriptor, and `Node` is the node where the instance is currently deployed.

Finally, application management agents can access data on the incoming requests targeting their application instance from neighbouring nodes, denoted as in

```
requests(ServiceInstanceId, Neighbour,
         RequestRate, LatencyToClient).
```

where `ServiceInstanceId` is the unique identifier of the managed application instance, `Neighbour` is the identifier of the last-hop neighbouring node from which requests reach the application instance, `RequestRate` is the average number of requests targeting the application instance in the last time period, and `LatencyToClient` is the end-to-end latency associated with the path traversed by the incoming requests, from the client to the considered application instance.

Facts of type service/4, node/2, serviceInstance/3 and requests/4 constitute the knowledge base accessible to each application management agent.[5] Figure 2a shows an example of knowledge base in which a video broadcasting service instance s1 (requiring 4 hardware units, serving at most 10 requests and tolerating an end-to-end latency of 25 ms) is running on node n42 featuring 10 free hardware units.

As sketched in Fig. 2b, the application instance s1 is receiving 2 requests from application clients c2 and c3, following two alternative routing paths (i.e. n41– n42 and n40– n41– n42), respectively. As illustrated in Fig. 2, the application management agent is only aware of last hop information about incoming requests, i.e. it ignores that requests from `n41` come from different routing paths. Requests from clients in same node as the service instance are assumed to have null latency and indicated by Prolog facts such as the

---

[4]For the sake of simplicity, we represent generic hardware units as integers, as in [15, 23]. Accounting for different types of resources (e.g. RAM, CPU, HDD) is a straightforward extension.

[5]More precisely, with reference to the semantics presented in Section 2.2, these Prolog facts implement the information on node idenfiers (n), sets of management agents (A_n), and available contextual data (K_n). The policies (p) of bacteria will be defined by the Prolog rules described in Section 3.3.

```
1  service(videoBroadcast, 4, 10, 25).
2  node(n42, 10).
3  serviceInstance(s1, videoBroadcast, n42).
4  requests(s1, self, 5, 0).
5  requests(s1, n41,  1, 15).
6  requests(s1, n41,  1, 16).
```

(a) Prolog facts at s1
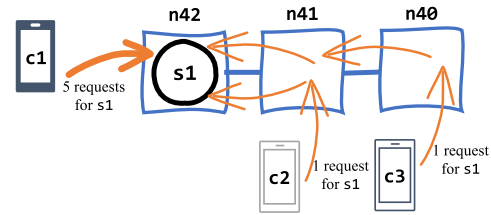
(b) Requests to s1

Fig. 2 Knowledge base of a MARIO$_2$ application management agent

one at line 4 in Fig. 2a, denoting direct requests of client c1 to s1.

3.2.2 Management Operations

The triggering of management operations described in Section 2 (viz. undeploy, migrate, replicate) can be declaratively specified by means of simple Prolog rules.

Each management operation and the associated TriggeringCondition are declared as a Prolog rule of the form:

```
operation(OperationId, ServiceInstanceId, TargetNode) :-
        TriggeringCondition.
```

where OperationId ∈ {undeploy, migrate, replicate}, ServiceInstanceId is the service instance to which the management rule applies, and TargetNode is the node targeted by the operation (set to self when the operation targets only the current deployment node). Note that we exploit Prolog's clause ordering to express (simple) priorities among management operations (defined by predicate operation/2).

Finally, we assume that application management agents can be aware of which operations that they requested during the last management cycles were inhibited by MARIO$_2$. This is done by including in their knowledge base facts denoted as in

```
inhibited(OperationId, ServiceInstanceId, TargetNode).
```

3.3 Declaring Management Policies

By means of four examples, we epitomise how it is possible to compactly declare management policies exploiting the ingredients described up to now in this section. Before starting, it is worth noting that management policies in MARIO$_2$ are actual Prolog

programs, which can be written – and debugged [24] – as such. Since Prolog relies upon a declarative programming paradigm, the management policies we will illustrate express their desired goals instead of the commands or steps that must be performed to reach out those.

Policy 1 (Workload-aware) – This policy is primarily intended to handle changes in the application workload by triggering suitable undeployment, migration and scaling operations. The main code[6] for *Policy 1* is listed in Fig. 3.

First, an undeploy operation is triggered when application instance Si is currently not receiving any client requests, i.e. when the knowledge base of the considered application management agent does not (\+ denotes Prolog negation) contain any requests/4 fact (lines 1–2).

Triggering a migrate operation requires to first collect all requests targeting application instance Si along with their last hop H and experienced latency L (line 4).[7] A migration of instance Si to node M triggers when:

(1a)   Si cannot handle the total amount of incoming requests as it exceeds the MaxRequestRate it can handle (line 6), **AND**

(2)   all the requests for Si have the same neighbour M (different from self, viz. dif(M, self)) as last hop in their routing path (line 7).

---

[7]Note that the call findall((H,R,L),requests(Si,H,R,L),Requests) in Fig. 3 (line 4) corresponds to retrieving the set Request of triples (H,R,L) such that requests(Si,H,R,L) holds.

```
1   operation(undeploy,Si,self) :-
2       \+ requests(Si,_,_,_).

3   operation(migrate,Si,M) :-
4       findall((H,R,L),requests(Si,H,R,L),Requests),
5       serviceInstance(Si, S, _), service(S,_,MaxRequestRate,_),
6       sumRequestRates(Requests,TotalRequestRate), TotalRequestRate>MaxRequestRate,    %(1a)
7       findall(K,member((K,_,_),Requests),Ms), sort(Ms,[M]), dif(M,self).             %(2)

8   operation(replicate,Si,M) :-
9       findall((H,R,L),requests(Si,H,R,L),Requests),
10      serviceInstance(Si, S, N), service(S,RequiredHW,MaxRequestRate,_),
11      sumRequestRates(Requests,TotalRequestRate), TotalRequestRate>MaxRequestRate,    %(1a)
12      mostRequestsFrom(Requests, M),
13        (
14         (dif(M,self)
15         ;
16         (M=self, node(N,AvailableHW,_), AvailableHW >= RequiredHW))
17        ).
```

Fig. 3 Policy 1: workload-aware management

Finally, a replicate operation of Si in node M triggers when the aforementioned condition (1a) holds. The chosen target node M is the node from which most requests are arriving to Si (line 12), and it can be one among the current deployment node (self) or its neighbours. It is worth noting that if self is the replication target, it must also feature enough hardware resources to host a new instance of the considered application (line 16).

Policy 2 (Latency-aware) – This policy is primarily intended to reduce the client-to-application end-to-end latency to keep it below the maximum value tolerated by the application. The main code for *P*olicy 2 is listed in Fig. 4.

The undeploy operation triggers under the same condition of *Policy 1*. Conversely, a migrate operation of instance Si to node M triggers when:

(1b)   Si does not satisfy the maximum latency constraint for all requests arriving from M (line 6), **AND**

(2)   all the requests for Si have the same neighbour M (different from self) as last hop in their routing path (line 7).

Finally, a replicate operation of Si in node M triggers when the aforementioned condition (1b) holds (line 11), and M is the neighbour of self from which most requests to Si not satisfying the maximum latency constraint are arriving (lines 12–13).

```
1   operation(undeploy,Si,self) :-
2       \+ requests(Si,_,_,_).

3   operation(migrate,Si,M) :-
4       findall((H,R,L),requests(Si,H,R,L),Requests),
5       serviceInstance(Si, S, _), service(S,_,_,MaxLatencyToClient),
6       member((_,_,Lat),Requests), Lat>MaxLatencyToClient,                % (1b)
7       findall(K,member((K,_,_),Requests),Ms), sort(Ms,[M]), dif(M,self).  % (2)

8   operation(replicate,Si,M) :-
9       findall((H,R,L),requests(Si,H,R,L),Requests),
10      serviceInstance(Si, S, _), service(S,_,_,MaxLatencyToClient),
11      member((_,_,Lat),Requests), Lat>MaxLatencyToClient,                % (1b)
12      findall((K,Rk,Lk),(member((K,Rk,Lk),Requests),Lk>MaxLatencyToClient),FilteredRequests),
13      dif(FilteredRequests,[]), mostRequestsFrom(FilteredRequests, M).
```

Fig. 4 Policy 2: latency-aware management

Policy 3 (Workload-aware and Latency-aware) – This policy is intended to both handle changes in the application workload (as per *Policy 1*) and to reduce the client-to-application end-to-end latency to keep it below the maximum value tolerated by the application (as per *Policy 2*). The main code is listed in Fig. 5.

The undeploy still triggers under the same condition of *Policy 1* and 2. With reference to properties (1a), (1b) and (2) described in *Policy 1* and 2, a migrate operation of instance Si to node M triggers when it holds ((1a) **OR** (1b)) **AND** (2). Particularly, when:

(1a)   Si cannot handle the total amount of incoming requests as it exceeds the MaxRequestRate it can handle (line 7) **OR**

(1b)   Si does not satisfy the maximum latency constraint for all requests arriving from M (line 9), **AND** (irrespectively of which between (1a) and (1b) holds, or if they both hold)

(2)   all the requests for Si have the same neighbour M (differing from self) as last hop in their routing paths (line 11).

Finally, a replicate operation of Si in node M triggers when condition (1a) (line 16) **OR** (1b) (line 18) hold. Then, in case (1a) holds and not (1b) (viz. the list of requests not satisfying latency constraints is empty), M is chosen to be the node from which most requests are arriving to Si (lines 20–22), and it can be one among the current deployment node (self) or its neighbours (lines 24–26) as in *Policy 1*. Otherwise, if (1b) holds, the target node M is the neighbour of self from which most requests to Si not satisfying the maximum latency constraint are arriving (line 30) as in *Policy 2*.

Policy 4 (Workload-aware and Latency-aware, with Memory) – This last policy, listed in Fig. 6, has the same objectives as *Policy 3*. However, it refines the behaviour of *Policy 3* by avoiding to trigger a migrate or a replicate to a node M if such operation has been inhibited by MARIO$_2$ (lines 4 and 14) during the last management cycles. This simple refinement of *Policy 3* allows the application management agent to find an alternative management operation to execute, thus

```
1   operation(undeploy,Si,self) :-
2       \+ requests(Si,_,_,_).

3   operation(migrate,Si,M) :-
4       findall((H,R,L),requests(Si,H,R,L),Requests),
5       serviceInstance(Si, S, _), service(S,_,MaxRequestRate,MaxLatencyToClient),
6       (
7        (sumRequestRates(Requests,TotalRequestRate), TotalRequestRate>MaxRequestRate)      %(1a)
8        ; % OR
9        (member((_,_,Lat),Requests), Lat>MaxLatencyToClient)                               %(1b)
10      ),
11      findall(K,member((K,_,_),Requests),Ms), sort(Ms,[M]), dif(M,self).                  %(2)

12  operation(replicate,Si,M) :-
13      findall((H,R,L),requests(Si,H,R,L),Requests),
14      serviceInstance(Si, S, N), service(S,RequiredHW,MaxRequestRate,MaxLatencyToClient),
15      (
16       (sumRequestRates(Requests,TotalRequestRate), TotalRequestRate>MaxRequestRate)      %(1a)
17       ; % OR
18       (member((_,_,Lat),Requests), Lat>MaxLatencyToClient)                               %(1b)
19      ),
20      findall((K,Rk,Lk),(member((K,Rk,Lk),Requests),Lk>MaxLatencyToClient),FilteredRequests),
21      (
22       (FilteredRequests=[], mostRequestsFrom(Requests, M),
23        (
24         (dif(M,self)
25         ; % OR
26         (M=self, node(N,AvailableHW,_),AvailableHW>=RequiredHW))
27        )
28       )
29       ; % OR
30       (dif(FilteredRequests,[]), mostRequestsFrom(FilteredRequests, M))
31      ).
```

Fig. 5 Policy 3: workload- and latency-aware management

```
1   operation(undeploy,Si,self) :-
2       \+ requests(Si,_,_,_).

3   operation(migrate,Si,M) :-
4       \+ inhibited(migrate,Si,M),                                           % memory
5       findall((H,R,L),requests(Si,H,R,L),Requests),
6       serviceInstance(Si, S, _), service(S,_,MaxRequestRate,MaxLatencyToClient),
7       (
8        (sumRequestRates(Requests,TotalRequestRate), TotalRequestRate>MaxRequestRate)    %(1a)
9        ;
10       (member((_,_,Lat),Requests), Lat>MaxLatencyToClient)                   %(1b)
11      ),
12      findall(K,member((K,_,_),Requests),Ms), sort(Ms,[M]), dif(M,self).     %(2)

13  operation(replicate,Si,M) :-
14      findall((H,R,L),(requests(Si,H,R,L),\+inhibited(replicate,Si,H)),Requests),    % memory
15      serviceInstance(Si, S, N), service(S,RequiredHW,MaxRequestRate,MaxLatencyToClient),
16      (
17       (sumRequestRates(Requests,TotalRequestRate), TotalRequestRate>MaxRequestRate)    %(1a)
18       ;
19       (member((_,_,Lat),Requests), Lat>MaxLatencyToClient)                   %(1b)
20      ),
21      %
22      findall((K,Rk,Lk),(member((K,Rk,Lk),Requests),Lk>MaxLatencyToClient),FilteredRequests),
23      (
24       (FilteredRequests=[], mostRequestsFrom(Requests, M),
25        (
26         (dif(M,self)
27         ;
28         (M=self, node(N,AvailableHW,_),AvailableHW>=RequiredHW))
29        )
30       )
31       ;
32       (dif(FilteredRequests,[]), mostRequestsFrom(FilteredRequests, M))
33      ).
```

Fig. 6 Policy 4: workload- and latency-aware management, with memory

avoiding to get stuck trying to request the same operation over and over, e.g. migrating/replicating instead to a node different from those that recently refused its requests.

4 Experimental Results

In this section, we first describe (Section 4.2) a lifelike use case that we devised to validate *Policies 1–4* presented in Section 3.3. Then (Section 4.3), we present the results of extensive simulations of the proposed policies over such scenario. The simulation were carried on with the YAFS simulator (Section 4.1). Videos of the executed simulation are online[8] and show how the simulated MARIO$_2$ system evolves over time while reacting to user movements as per *Policies 1–4*.

4.1 YAFS Simulation

All simulations were carried on with YAFS [16],[9] a Python-based, lightweight and easy to configure, discrete event simulator (DES) for Fog computing scenarios.

Each application management agent is modelled by a DES process that runs in a specific node of the topology. Each DES agent also represents a unique service instance, and it gathers its local facts (node, service, serviceInstance and requests) from its knowledge base K_n as shown in Fig. 2. The DES agent periodically calls a Prolog execution to evaluate these facts with the rules assigned and it sends its requested operation (viz., a tuple $\langle b, n, op \rangle$ with op $\in$ {undeploy, migrate, replicate} defined in Section 2.2) to MARIO$_2$.
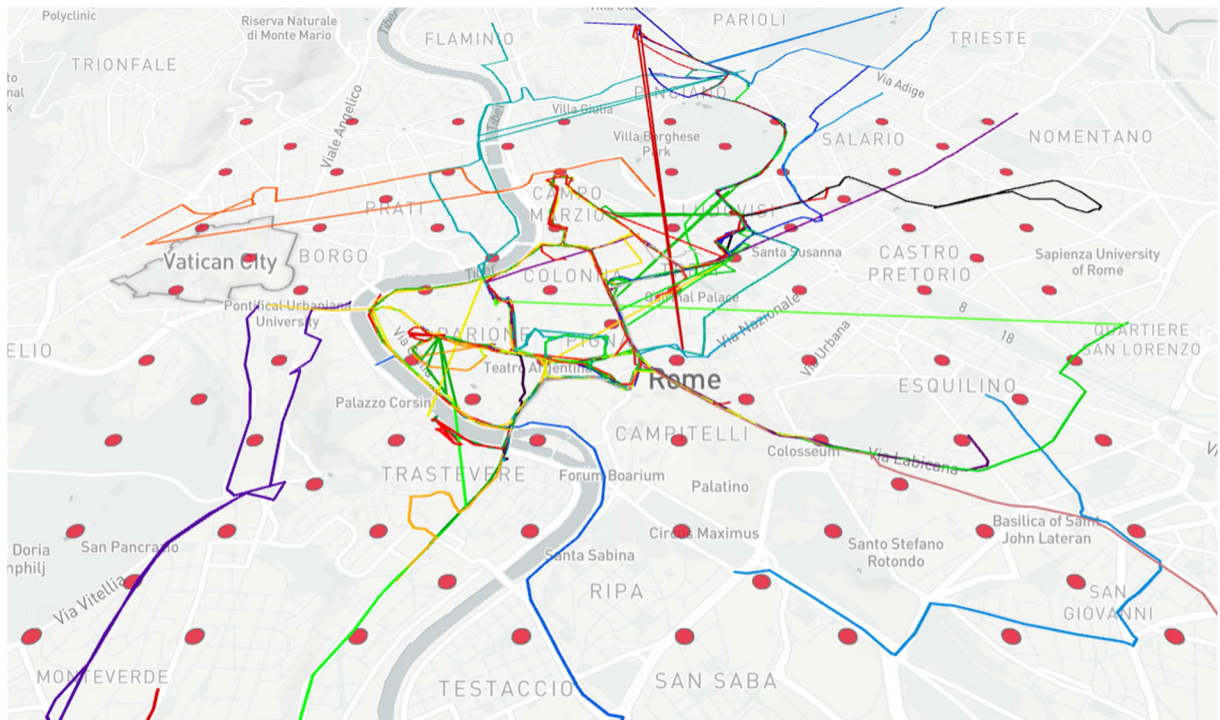

Fig. 7 Taxi traces and Fog nodes projected on the map of Rome

MARIO$_2$ is in turn another DES process that periodically evaluates all operation requested by the DES agents. MARIO$_2$ chooses the (last) operation requested by each agent and it either performs or inhibits the required operation. When MARIO$_2$ finishes evaluating all the requests, the routing network process of linking requests between users and service instances is updated, and new service instances or new allocations are considered. The coherence of the model is ensured by the fact that each DES process is an atomic process inside the simulator.

4.2 Experiments Setup

To assess *Policies 1–4*, we have defined a lifelike scenario based on taxi mobility in the city of Rome, Italy. We have considered a region of Rome of 4×4 km^2, and we have simulated 51 users based on actual taxi traces taken from [17]. The considered area of Rome and the user mobility traces are sketched in Fig. 7.

Since, to the best of our knowledge, there is not publicly available data on the location of the cellular antennas in Rome, we uniformly distributed 64

Access Points (APs) for the experiments using a grid distribution with AP-to-AP distance of 500 m. As shown in Fig. 8, we consider a tree-based infrastructure with a total of 85 nodes of heterogeneous capacity, where the APs are the edge nodes.
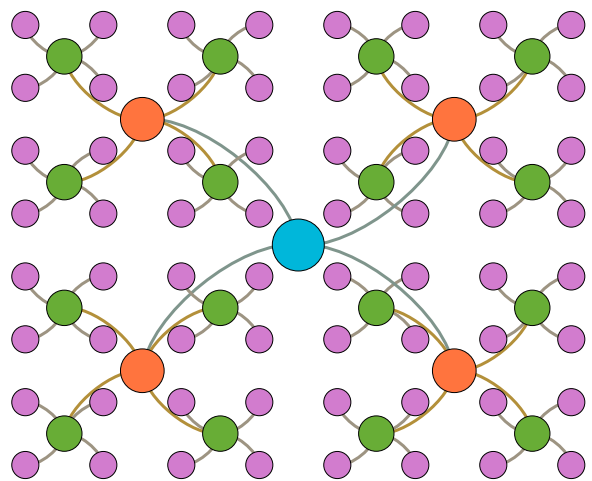
Fig. 8 Infrastructure topology

The nodes are distributed in a 4-tiered hierarchy consisting of:

– 1 central Cloud (denoted by an azure circle) with virtually unbounded hardware and 3 ms link latency to the nodes in the lower tier,
– 4 mini-datacentres (orange circles) with 9 hardware units and 2 ms link latency to the nodes in the lower tier,
– 16 edge nodes (green circles) with 6 hardware units and 3 ms latency to the AP nodes, and
– 64 APs (purple circles) with 1 hardware unit.

Realistically, we assume that moving users always connect to the closest AP and we track the sequence of AP handovers (i.e. users changing their AP connection). These handovers are indeed responsible for the dynamic conditions of the experiments. Users are randomly split in two groups – each group using different applications – keeping the same average number of handovers in both groups.

For the purposes of the experiments, we have considered six different applications declared as listed in Fig. 9.

The six applications are divided into two types:

– *workload-sensitive* (WS, app1–app3), capable of handling a low request rate (5), and
– *latency-sensitive* applications (LS, app4–app6), needing to have very low end-to-end latencies from their clients (viz. 1 ms).

Under the aforementioned conditions, WS applications are expected to move closer to edge nodes when *Policy 1* (workload-aware) and *Policy 3* and *Policy 4* (workload- and latency-aware) are used, and to remain in the cloud when *Policy 2* (latency-aware) is used. On the other hand, LS applications are expected to move

closer to edge nodes when *Policy 2* (latency-aware) and *Policy 3* and *Policy 4* (workload- and latency-aware) are used, and to remain in the cloud when *Policy 1* (workload-aware) is used.

We assume that each user requests only one application and that half of the users use WS applications, and the other half use LS applications. We also take a uniform distribution of users' request rates, with each user sending a request every 40 units of simulation time. Each simulation is executed for 52,000 simulation units, because of the 25 user movements that are updated every 2,000 simulation units. The simulator also makes the state of the scenario evolve by changing the AP associated to each users as the simulation goes forward, simulating a total of 81 AP handovers. Figure 10 shows the initial deployment of the applications (denoted by numbered squares in the central Cloud node) and the connections of the users to the APs (denoted by semicircles below the nodes).

Four experiments were designed and carried out by assigning different management policies to different application types. In the initial state of the simulation, all the applications only have one instance deployed at the central Cloud node.

4.3 Results and Discussion

The objective of our simulations is to assess both dynamic (i.e. transient) and static (i.e. steady) application deployment conditions. On one hand, they aim at assessing how promptly the proposed declarative, fully-decentralised management policies can respond to application users' mobility, to workload variations and to violations of latency constraints required by running application instances. On the other hand, they aim at assessing how the policies converge to a steady

```
% service(ServiceId, RequiredHW, MaxRequestRate, MaxLatencyToClient).

%% workload-sensitive %%
service(app1, 1, 5, 10).
service(app2, 1, 5, 10).
service(app3, 1, 5, 10).

%% latency sensitive %%
service(app4, 1, 40, 1).
service(app5, 1, 40, 1).
service(app6, 1, 40, 1).
```
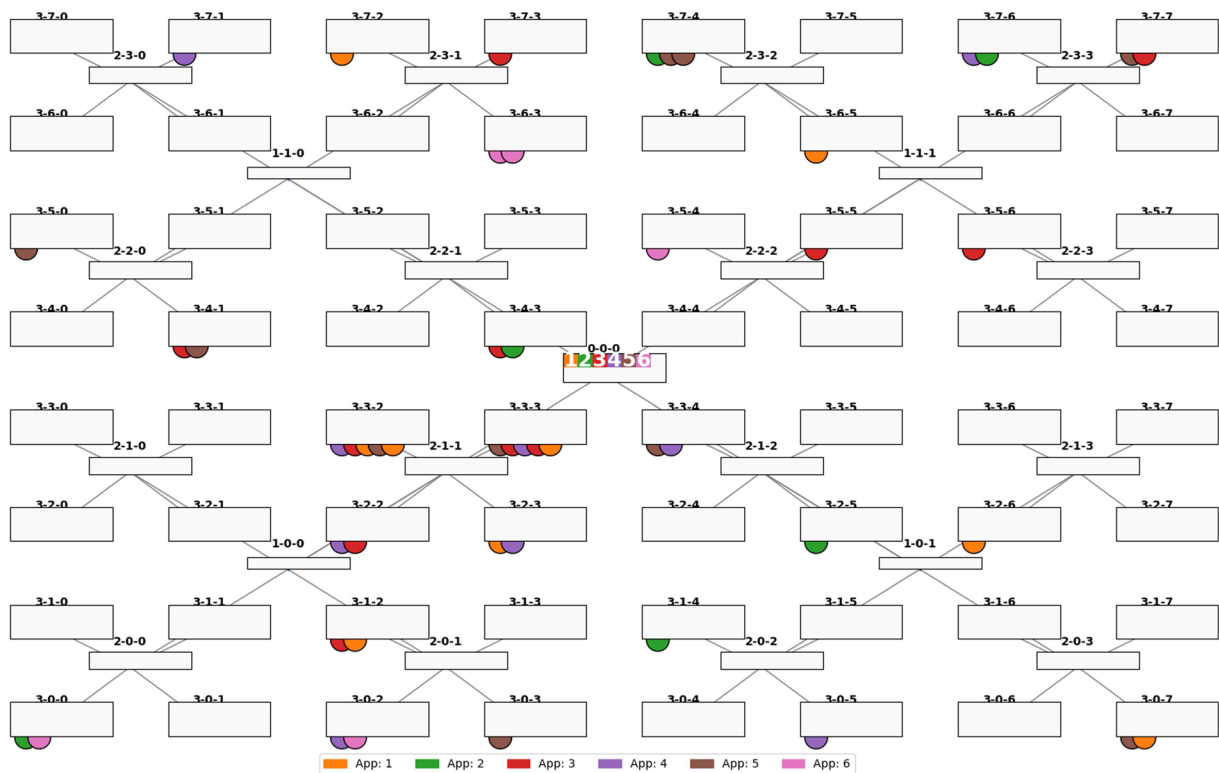
Fig. 9 Example applications

Fig. 10 Initial experiment setup

allocation of the applications over the available Fog infrastructure if no changes occur for a sufficiently long period of time.

As mentioned before, changes in the considered scenario are produced by the movements of the users. Every time a user changes the AP it connects to:

1. a number of migrate, replicate or undeploy operations trigger, and
2. after a finite number of management cycles, if no other AP handover happens, the instances affected by users' movements do not require further actions, or they get rejected (viz. *inhibited*) by the node manager to which they are directed.

To illustrate such behaviour, we first present a set of plots where the number of current instances in the system and the type of operation selected by each of them is shown for each evaluation of the management rules.[10] Additionally, the number of user handovers

---

[10]For the sake of the analysis, the plots also represent instances that do not require any action. These cases are represented in orange and named nop, to means that no action/operation is requested by a given service instance.

between two evaluations of the management rules is also represented. From the analysis of these plots, we can observe the number of MARIO$_2$ executions that is needed after the user movements to achieve a steady allocation of the service instances. This is represented by the number of periods including migrate, replicate or undeploy operations after a user movement. If there is no period without these types of operations between user movements, this means that application management agents are still *searching* for a suitable allocation of their service instance, i.e. a temporarily steady allocation.

Policy 1 in Action – The bar plots in Fig. 11 show the type of operations executed by the application instances along the simulation cycles, when *Policy 1* is enforced by application management agents. Each bar is divided according to the number of actions of each type that has been triggered by running application managers. As *Policy 1* is workload-aware, new instances of the application are generated when the already running instances are not able to handle all incoming requests.
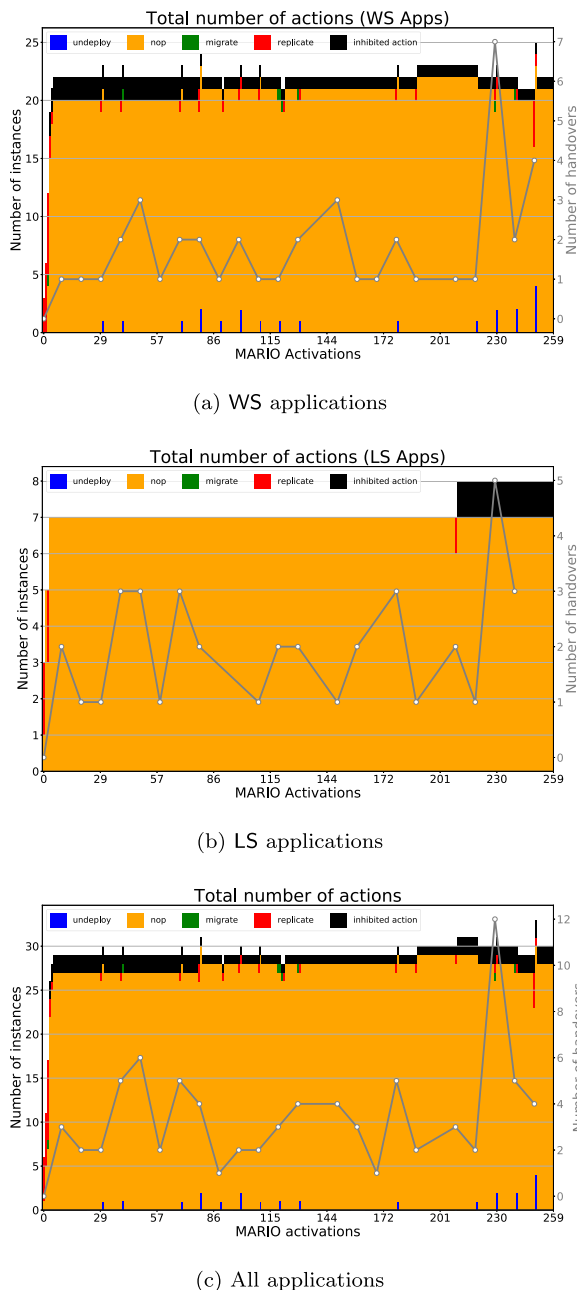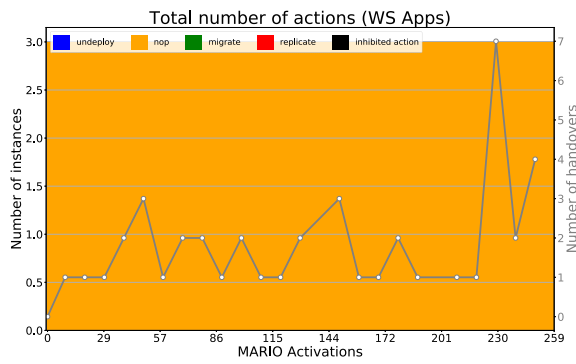
(a) WS applications

(b) LS applications

(c) All applications

Fig. 11 Experiments with *Policy 1*

It is worth noting that WS apps were defined with a low capacity of request response and, on the contrary, the LS were defined with a low capacity of request response. From Fig. 11a and b, we observe that the total number of application instances in the experiment settles at around 20-25 for the WS applications (with all types of operation along the simulation) and at around 7 for the LS applications with mainly

only nop and *inhibited* operations (orange and black). The observed emerging behaviour of the WS apps (Fig. 11a) corresponds to the case in which the initial instances in the central Cloud cannot handle the total number of requests. Hence, they replicate and migrate closer to the users as per *Policy 1*. On the contrary, the behaviour observed in the case of LS apps (Fig. 11b) corresponds to the case that the initial single instances of the applications are enough to handle all the request in the system (even though constantly violating the latency requirements). Consequently, the LS applications do not replicate, neither migrate closer to users as per *Policy 1*.

From the plot that aggregates the operations from both sets of applications (Fig. 11c), although some applications instances are continuously blocked (corresponding to the *inhibited* operations), the system achieves a steady allocation after each set of user movements (represented with the dots of the blue line plot). Indeed, user movements are followed by a small number of management cycles where undeploy, migrate, and replicate operations take place. The cycles needed to reach a temporarily steady condition settle on average between 1 and 2, which translates into the ability of $MARIO_2$ to promptly react to user movements. The *inhibited* actions indicate that, although the allocation is stable, there are some applications that are not yet running in suitable nodes. They are trying to execute migrate or replicate operations but the node manager they are demanding to perform the operations rejects their request (because the node does not feature sufficient free hardware to host them).

Policy 2 in Action – Fig. 12 shows the result when policy *Policy 2* is used across all applications, i.e. with the goal of migrating and replicating instances in a latency-aware manner. Remember that WS were defined with a relaxed latency constraint which does not trigger them to move away from the central Cloud (as per the initial setup), even though they are experiencing overloading. On the contrary, the LS set of applications has a tighter latency constraint, which forces them to move as close as possible to edge nodes as per *Policy 2*. Indeed, as per Fig. 12a, the three application instances of type WS are never replicated – if needed – nor migrated only incurring in nop operations. This brings to zero the number inhibited operations, as less resources are consumed in resource-constrained node at the edge of the network.
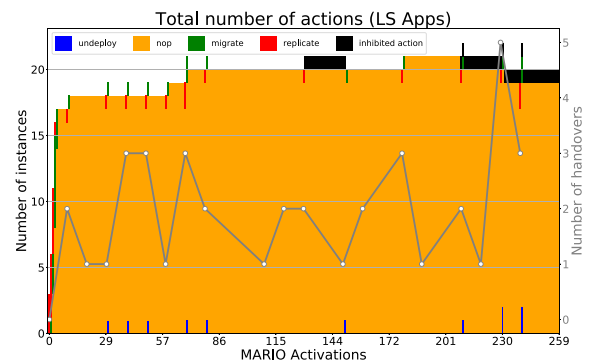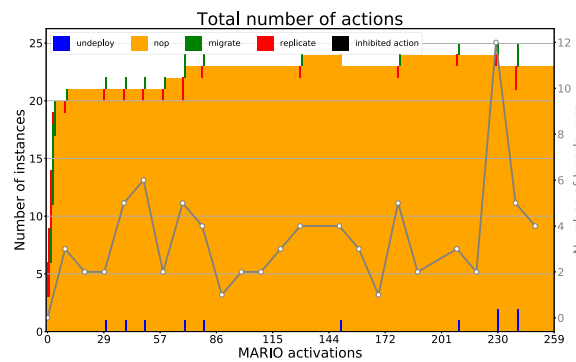
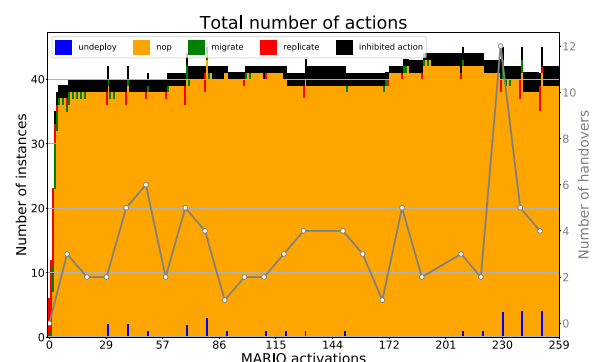(a) WS application s
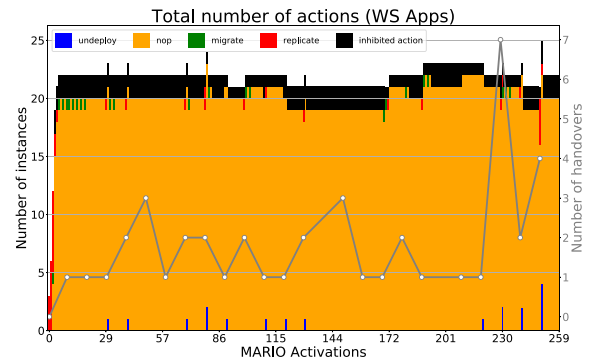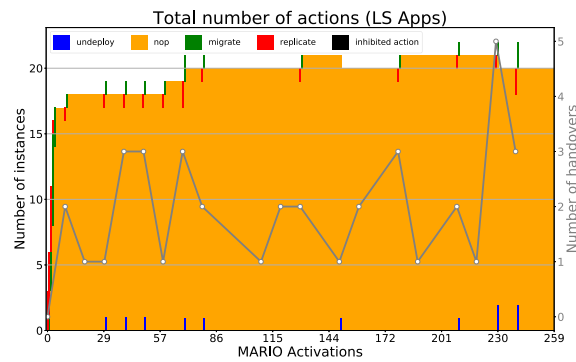
(b) LS applications

(c) All applications

Fig. 12 Experiments with *Policy 2*

(a) WS applications

(b) LS applications

(c) All applications

Fig. 13 Experiments with *Policy 3*

As expected, Fig. 12a shows instead how LS application instances increase up to 21 throughout the simulation in order to get closer to their users and satisfy their latency constraints.

Again, as illustrated in Fig. 12c, only 1 or 2 management cycles suffice to reach a temporarily steady allocation in response to user movements. Differently from the experiments with *Policy 1*, in these settings,

all application instances achieve their suitable allocation at every step of the simulations as no inhibited operation is observed.

Policy 3 in Action – Fig. 13 shows the results for the third experiment, where *Policy 3* is enforced across all applications. It is worth recalling that this policy

is both workload-aware and latency-aware. Consequently, the applications from both set should try be replicated and migrated closer to the users. As shown in Fig. 13a and b, both types of applications scale up to 20 instances and more as per *Policy 3*. Naturally, as both types of applications are trying to scale, the number of inhibited actions increases in the considered scenario. It is worth noting, however, that as in the experiments with *Policy 1* only (Fig. 11), operations on WS applications are inhibited more often than operations on LS applications (Fig. 13c). This is due to the fact that trying to replicate on edge nodes is not always possible due to their limited resources. On the other hand, once an application of type LS reaches a node from which it can guarantee the required latency constraints, it will remain there all through until its users move, thus triggering either a migrate or an undeploy operation (Fig. 13a).

As illustrated by Fig. 13c, despite more application instances need to adapt their allocation in response to user movements, the management cycles needed to react to changes is still reasonably small, being bound by 2 on average and by 6 in the worst-case (which only happens once in the simulation).

Policy 4 in Action – The last experiment corresponds to the use of *Policy 4*, which extends *Policy 3* by avoiding that a recently inhibited management operations is tried again. For the sake of the experiments, we forbid retrying operations that were rejected in the previous management cycle (i.e. we keep in the knowledge base of each management agent only the inhibited/3 facts related to the last policy evaluation) and in the previous 10 management cycles.

Even with a small memory of inhibited actions, the general behaviour of *Policy 4* is very similar to the one of *Policy 3* but it shows a 23% reduction of the number of inhibited operations across all applications (Fig. 14). Indeed, the behaviour of WS applications plot in Fig. 14a shows that inhibited agents can almost always find an alternative operation in the next management cycle (differently from *Policy 3*, Fig. 13a). Factually, we can observe that there are fewer consecutive *inhibited* actions throughout the simulation. On the other hand, the behaviour of LS applications is practically unchanged as shown in Fig. 14b. As in all the previous experiments, the management operations needed to react to changes the applications are bound to a small constant, viz. 2 on average (Fig. 14c).
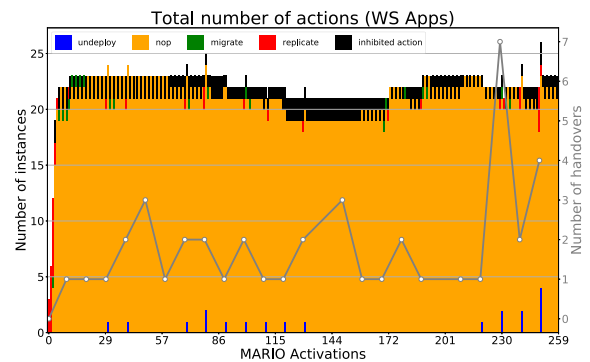
(a) WS applications
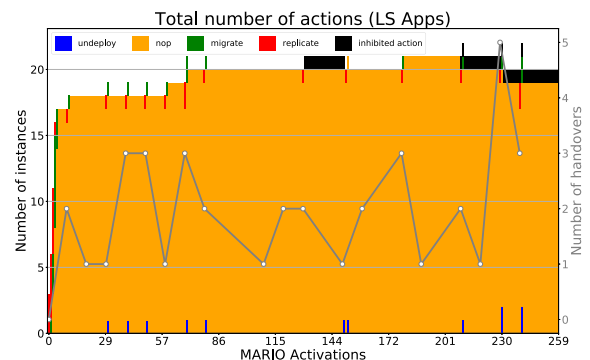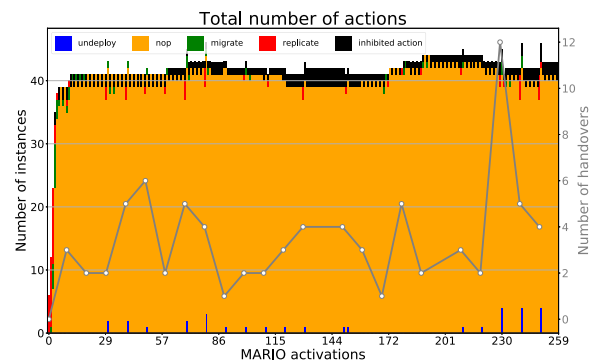
(b) LS applications

(c) All applications

Fig. 14 Experiments with *Policy 4* (memory size 1)

As expected, increasing the memory size to the last 10 management cycles further improves these results (Fig. 15). Indeed, WS applications are now capable of always finding a suitable alternative to the previously inhibited operation in the consecutive management cycle, thus reducing the period needed to react to changes (Fig. 15a). Similarly, LS applications incur
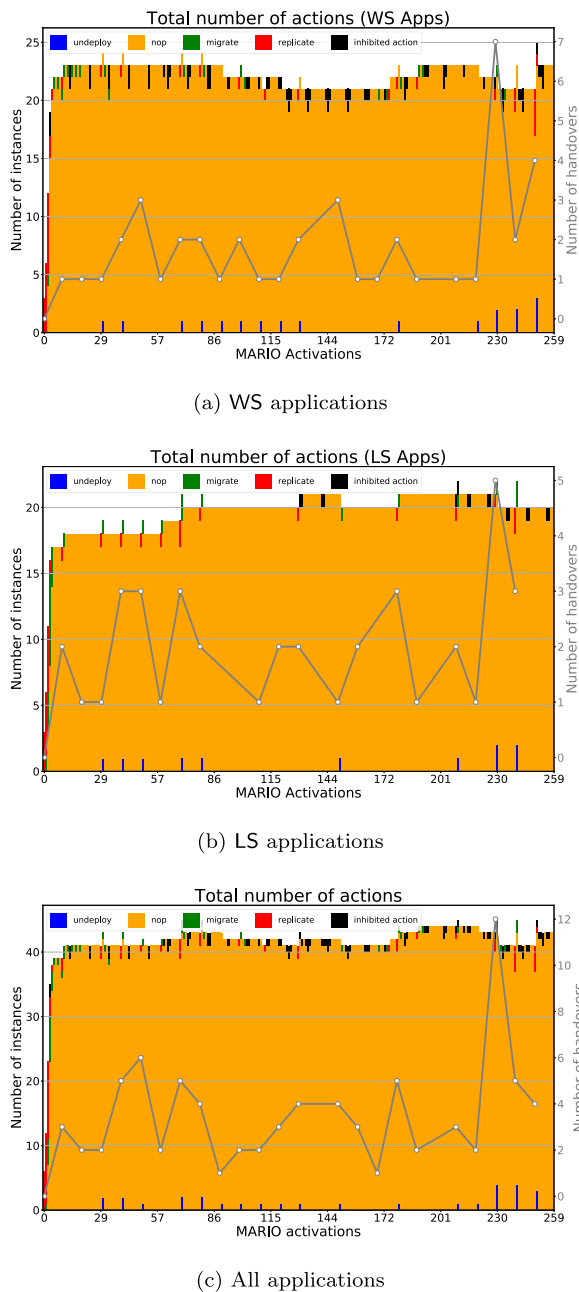
(a) WS applications

(b) LS applications

(c) All applications

Fig. 15 Experiments with *Policy 4* (memory size 10)

in a negligible number of inhibited actions throughout the simulation, viz. 16 throughout the simulation (Fig. 15b). Such improved emerging behaviour of the system is also shown by Fig. 15c, aggregating on both WS and LS applications. Overall inhibited actions are reduced by more than 80% in this last scenario, with respect to *Policy 3*.

Correlating Management Operations and User Movements – Additionally, we have analysed how the number of user movements influences the number of migrate, replicate or undeploy operations. Expectedly, the higher the number of user movements, the higher the number of operations that change the allocation of the services. We have measured this influence by calculating the Pearson's correlation coefficient (r) between the series that counts the number of user movement for each period and the series that counts the number of migrate, replicate or undeploy operations in those same periods. These correlation evaluations have been performed independently for each set of applications (WS and LS) and for the aggregation of both sets. Figures 16, 17 and 18 plot the two series for each case, and the resulting Pearson's correlation coefficient.

Generally speaking, there is a strong correlation between the movement of the users and the number of management operations performed in the system. However, such correlation is only present when management policies try to allocate applications closer to the users. This is the case, for example, of WS applications managed via *Policy 1* ($r = 0.597$, Fig. 16) and of LS applications managed via *Policy 2* ($r = 0.985$, Fig. 17). On the contrary, the correlation is very low for the application set that satisfied the constraints by just being allocated further from the users, for example in the Cloud, requiring almost no operation. This is the case of LS applications managed via *Policy 1* and of WS applications managed via *Policy 2*. Notice that correlation coefficient is not indicated in Fig. 17b. This is because one of the series has always value 0, so coefficient cannot be calculated.

Management Operations per User Movement – We measured the cost, in terms of number of operations, generated by each user movement. To do that, we have calculated the average number of operations per user movement by dividing the total number of operations of one type by the total number of user movements. We have limited this analysis to the case of the operations that generates changes in the allocation of the services (migrate, replicate or undeploy operations). Table 1 shows the values separated for policies, operations and applications types.
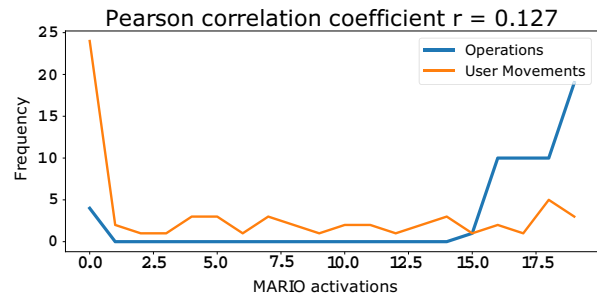
It is worth noting that the number of operations to react to changes and adapt the application deployment is very low on average, i.e. ≤ 1 for all policies.
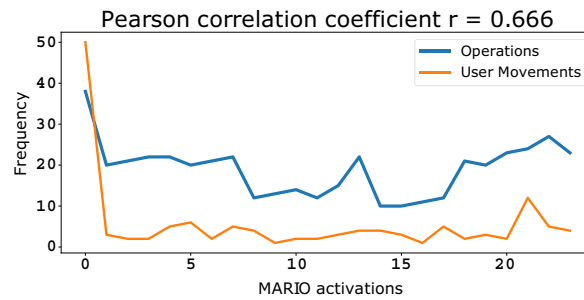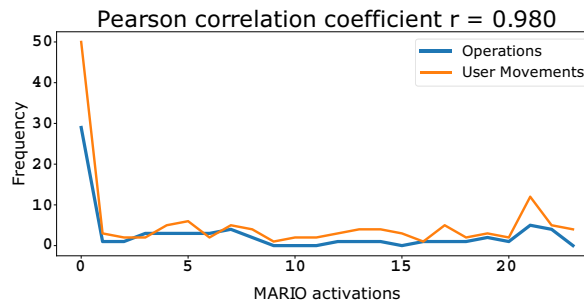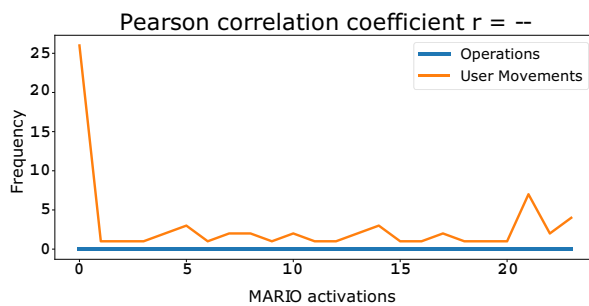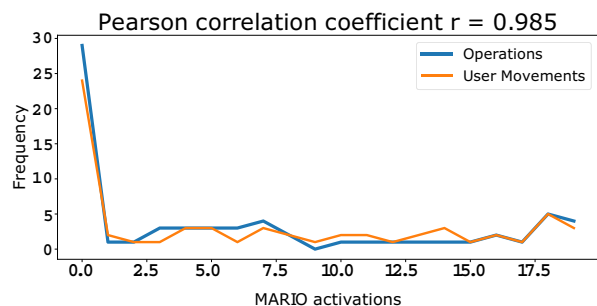
(a) All applications
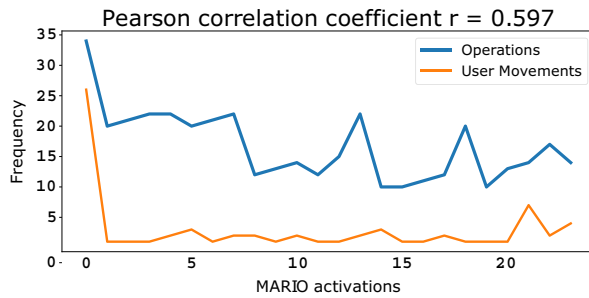
(b) WS applications

(c) LS applications

Fig. 16 Pearson correlation coefficient for *Policy 1*

(a) All applications

(b) WS applications

(c) LS applications

Fig. 17 Pearson correlation coefficient for *Policy 2*

(a) All applications
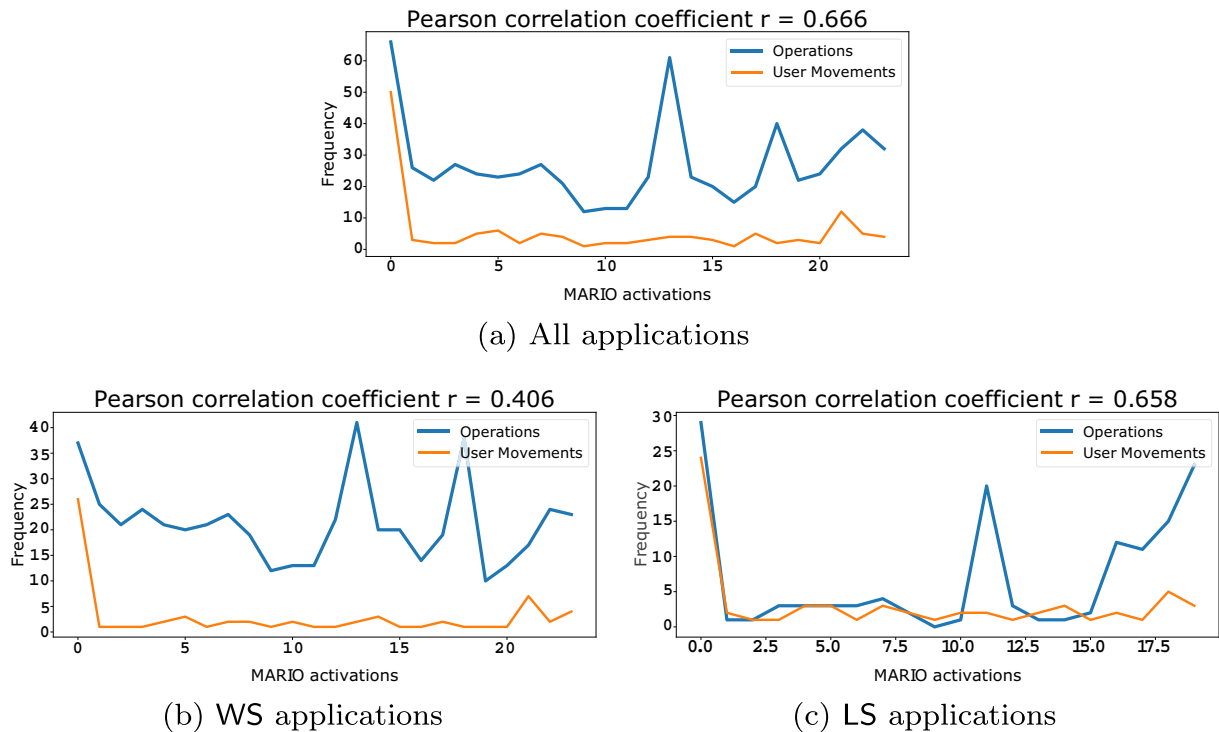

(b) WS applications

(c) LS applications

Fig. 18 Pearson coefficient for *Policy 3*

Consequently, as aforementioned, the cost in terms of management operations is indeed very low and permits fast convergence in practice for the considered lifelike scenario.

Policies and Fog System Performance – Each rule is designed to compensate or balance applications in terms of their sensitivity to latency and workload. We have carried out an analysis of the average response time of each type of application and also a calculation of the level of overload due to the ratio of requests that each application can attend. More precisely, service usage is measured as the ratio between the requests received and the total number of requests that can be served by a service.

The average values of response times and service usages are shown in Table 2. The first two rows of Table 2 highlight how *Policy 1* gives preference to workload-sensitive applications, while *Policy 2* gives preference to latency-sensitive apps. As for response time, *Policy 1* prioritizes WS apps and its response time is 3.104 ms, while these applications suffer a response time increase of 407% with *Policy 2*. LS apps that benefited from *Policy 2* have an improvement of 227%. *Policy 3* and *Policy 4* compensate for both latency and load characteristics. Both types of applications get fairly similar response times. It is worth noting that a balance between both application types does not suppose the best assumption and therefore the LS apps do not obtain

Table 1 Average number of actions by movement

Action	Policy 1			Policy 2			Policy 3			Policy 4		
	All	WS Apps	LS Apps	All	WS Apps	LS Apps	All	WS Apps	LS Apps	All	WS Apps	LS Apps
Replicate	0.548	0.930	0.128	0.341	0	0.717	0.792	0.860	0.717	0.829	0.906	0.743
Migrate	0.097	0.186	0	0.341	0	0.717	0.621	0.534	0.717	0.609	0.534	0.692
Undeploy	0.256	0.488	0	0.134	0	0.282	0.353	0.418	0.282	0.378	0.441	0.307

Table 2   Experiment performance results

Experiment	Avg. Response Time (ms)		Service usage	
	WS Apps	LS Apps	WS Apps	LS Apps
Policy 1 *Workload-aware*	3.104	8.611	1.117	0.592
Policy 2 *Latency-aware*	15.757	2.628	8.750	0.302
Policy 3 *Workload- and Latency-aware*	2.727	2.716	1.165	0.318
Policy 4 *Workload- and Latency-aware, with Memory*	2.590	2.652	1.131	0.303

the best result compared to *Policy 2*. Furthermore, *Policy 4* also represents an improvement compared to *Policy 3*.

In relation to the service usage, the values of *Policy 1* and *Policy 2* are again appreciated in the two types of applications. For example, the WS Apps with *Policy 2* is highly overloaded with a value of 8.750 and this affects the response time. On the contrary, *Policy 2* compensates LS apps by generating more instances than in *Policy 1* and thus usage service level is reduced and consequently response time improves. *Policy 3* and *Policy 4* balance the service usage of both types of applications compared to the previous two policies.

4.4 Scalability

Generally speaking, a solution is considered to be scalable if the complexity of the problem does not increase when its size does (e.g. number of infrastructure nodes, number of services, number of users).

In comparison with centralised solutions, in MARIO$_2$ each application instance only considers locally available monitored data about the neighbour infrastructure and only data about user requests directly targeting it.[11] This considerably reduces the amount of information that agents have to process to make informed management decisions at runtime. While the management-related problems that agents need to solve are still worst-case exp-time (i.e. placement, sca-

ling, migration), solving them over small instances – with few 10s of nodes – is always (exhaustively) feasible as compared to solving them in large-scale settings – with 100s of nodes [25]. Additionally, it is important to note that the agents evaluate the management rules in parallel due to their distributed and decentralized execution.

Summing up, we can say that MARIO$_2$ succeeds in mastering the exp-time complexity of large-scale Fog infrastructures in that:

– The time needed by an infrastructure to "act" (by accepting/rejecting bacteria's requests) is independent of the number of bacteria in the nodes. All bacteria generate their proposed actions in parallel, and each proposal in this stream is accepted/rejected by the infrastructure nodes in negligible time.
– The time needed by the infrastructure nodes to "act" (by accepting/rejecting bacteria's requests) does not change if we add an arbitrary number of nodes to the infrastructure (but for the case of fully connected infrastructures, which are however not feasible in the Fog domain).

5 Related Work

Much literature has focussed on the placement, orchestration and management of application services to physical servers in Cloud datacentres (e.g. [26, 27]). However, only few authors employed declarative programming (e.g. [28, 29]) or decentralised approaches

---

[11]Note that the number of neighbours of each node (5 in our experiments) depends on the degree of the network graph, not on its size.

(e.g. [30, 31]), and none combined both. Moreover, managing applications over the Cloud-IoT continuum introduces new peculiar challenges related to infrastructure scale, lower resource availability, churn and heterogeneity, and to the specificity of IoT application needs, which were rarely considered in Cloud-only scenarios [32, 33].

Many authors tackled the problem of (sub-) optimally placing and managing services along Cloud-IoT infrastructures. However, few approaches are decentralised. We refer the readers to recent surveys [5–7] for further details on these topics, and we hereby focus on proposals tackling decentralised and/or declarative approaches to realise next-gen orchestration of Fog applications.

Aiming at collaborative distributed application deployments based on contextual data, Colistra et al. [34] proposed a middleware and a consensus protocol (via broadcast or gossip communication) to achieve decentralised management according to a single non-declarative policy targeting load-balancing. Further work on $MARIO_2$ could investigate embedding a consensus protocol inspired from theirs, to decide when management operations involve more nodes. Mennes et al. [35] proposed the GRECO framework, which relies upon distributed genetic algorithms to determine application placements which are likely to resist infrastructure churn, in a failure-resilient manner, by also considering backup application replicas. However, despite the algorithm runs on distributed workers, decision-making still relies upon global knowledge on the available infrastructure.

More recently, Guerrero et al. [36] proposed a decentralised placement and migration strategy to reduce client-service hop distances. Their work aims at keeping services with lower request rates farther from their users, and services with higher request rates closer to their users. Experimental assessment via simulation has shown promising results in terms of latency reduction and bandwidth usage, with respect to a centralised policy. However, a single non-declarative policy – which can be expressed also in $MARIO_2$ – is used for managing applications.

Considering horizontal and vertical application scaling operations, inspired from biological systems, Herrera and Moltó [37] proposed a platform for self-managing containers to optimise load-balancing in a fully decentralised manner. Their approach leverages a single non-declarative policy based on a normalised score accounting for the usage of different resources (i.e. CPU, RAM, I/O), thus disclosing access to those information differently from $MARIO_2$.

Similarly, focussing on application scaling, Rossi et al. [38] extended Kubernetes decentralised orchestration with self-adaptation and network-aware placement capabilities using reinforcement learning. However, decisions are still made by a centralised control entity relying on a global view of the system.

Declarative approaches have been proposed in the past to improve network management (e.g. [39]) or to express and document management policies in centralised systems [40]. Only recently, few works exploited logic programming to incrementally determine eligible application placements in Fog scenarios [9], to place VNF chains and steer traffic across them [41], and to enable the assessment of the security and trust levels of different placements by means of algebraic structures [42]. From a different perspective, [43] employed a declarative approach to describe application requirements based on OpenTOSCA and to automatically deploy applications on different platforms.

Recently, Casadei et al. [44] and Pianini et al. [45] proposed a declarative approach to service coordination based on *aggregate computing* [46], i.e. *Self-organising Coordination Regions* (SCR). SCR aims at managing and coordinating opportunistic resources via a hybrid centralised/decentralised solution, by relying on a self-organising peer-to-peer architecture to handle churn and mobility. Differently from $MARIO_2$, in SCR, few Leader nodes make placement decisions, and application code need restructurings as per the aggregate computing paradigm.

From the point of view of simulating Fog environments, besides the YAFS simulator [16] – which we used to assess $MARIO_2$ – many proposals exist for simulating management policies in Cloud-IoT scenarios [47], e.g. iFogSim [48], EdgeCloudSim [49], and FogDirSim [50]. All those simulators, despite potentially supporting decentralised management policies, were mainly used to assess the performance of simple centralised policies (e.g. random, first-fit, best-fit) reasoning on the conditions of the whole system. Finally, tackling problems other than decentralised application management, bacteria-inspired solutions – such as $MARIO_2$ – have shown promising results in handling the task allocation for systems with multiple unmanned aerial vehicles [51], Cloud security [52],

software-defined network communication [53] and self-organising 5G communication networks [54].

Summing up, to the best of our knowledge, no previous work targeting Fog application management exploited a bacteria-inspired, fully decentralised and declarative approach, which permits expressing and flexibly enforcing *ad-hoc* context-aware management policies for different (classes of) applications.

6 Concluding Remarks

In this article, by substantially extending our preliminary previous work [15], we have presented $MARIO_2$: a bacteria-inspired, declarative and fully-decentralised solution for managing next-gen applications in opportunistic Fog computing settings, when it is not possible to adopt centralised solutions.

After defining an operational semantics of our management solution based on three simple operations (viz. undeploy, migrate, replicate), inspired from the emerging behaviour of bacteria colonies, we have shown how to use them to declare four different management policies by means of an open-source Prolog prototype of the proposed solution. We have shown how $MARIO_2$ allows to easily and declaratively specify *ad-hoc* management policies for different (classes of) applications that might want to promptly scale at increasing workload or to enforce latency constraints. Finally, we have assessed the proposed management policies over a lifelike scenario which considered clients mobility as per real traces of taxi drivers in Rome, Italy. Experimental results validate the proposed management policies and show that $MARIO_2$ is capable of promptly reacting to changes (i.e. users movements, consequent workload variations) within a few evaluations of the management policies, viz. 1 on average.

As application management agents are fully decentralised and run independently from each other, $MARIO_2$ can scale independently from the infrastructure size and avoids single points of failure that are typical of centralised approaches. Factually, the crash of a node hosting instances of a certain application does not prevent its other instances from replicating so to compensate for the unavailable ones. This, along with the bacteria-inspired behaviour, makes $MARIO_2$ self-organising, hence more resilient to user/device mobility and churn in general, which are typical of opportunistic Fog computing settings [11].

Last but not least, $MARIO_2$ allows users to specify *ad-hoc* policies for different (classes of) applications, as epitomised in our examples targeting latency-aware and workload-aware applications. As showcased in our experiments, such different dedicated policies can help different applications to optimise their own objective metrics. Along with possible extensions to the information available to $MARIO_2$ agents (e.g. security, costs), this permits to flexibly accommodating a range of diverse and ever-changing application needs in Cloud-IoT scenarios.

As future work, we intend to:

– extend the set of management operations featured by $MARIO_2$, inspired from other bacteria behaviour (e.g. spore-formation, evolution), so to define new management operations (e.g. service adaptation depending on target node as in Osmotic computing [2]),
– include other dimensions to the modelling of $MARIO_2$ such as bandwidth constraints, security policies or operational costs, and more complex priority relations among management operations, to drive management decisions,
– implement and assess $MARIO_2$ within an existing orchestration platform (e.g. Kubernetes) to test its functioning over Fog testbeds.

References

1. Habibi, P., Farhoudi, M., Kazemian, S., Khorsandi, S., Leon-Garcia, A.: Fog computing: a comprehensive architectural survey. IEEE Access **8**, 69105–69133 (2020)

2. Villari, M., Fazio, M., Dustdar, S., Rana, O., Jha, D.N., Ranjan, R.: Osmosis: The osmotic computing platform for microelements in the cloud, edge, and internet of things. Computer **52**(8), 14–26 (2019)

3. Pham, Q.-V., Fang, F., Ha, V.N., Piran, M.J., Le, M., Le, L.B., Hwang, W.-J., Ding, Z.: A survey of multi-access edge computing in 5g and beyond: fundamentals, technology integration, and state-of-the-art. IEEE Access **8**, 116974–117017 (2020)

4. Filali, A., Abouaomar, A., Cherkaoui, S., Kobbane, A., Guizani, M.: Multi-access edge computing: A survey. IEEE Access (2020)

5. Brogi, A., Forti, S., Guerrero, C., Lera, I.: How to place your apps in the fog: state of the art and open challenges. Softw. Pract. Exp. **50**(5), 719–740 (2020)

6. Mahmud, R., Ramamohanarao, K., Buyya, R.: Application management in fog computing environments: a taxonomy, review and future directions. ACM Comput. Surv., **53**(4) (2020)

7. Vaquero, L.M., Cuadrado, F., Elkhatib, Y., Bernal-Bernabe, J., Srirama, S.N., Zhani, M.F.: Research challenges in nextgen service orchestration. Future Gener. Comput. Syst. **90**, 20–38 (2019)

8. Ghobaei-Arani, M., Souri, A., Rahmanian, A.A.: Resource management approaches in fog computing: a comprehensive review. J. Grid Comput. **18**(1), 1–42 (2020). https://doi.org/10.1007/s10723-019-09491-1

9. Forti, S., Brogi, A.: Continuous reasoning for managing next-gen distributed applications. In: ICLP Technical Communications 2020, ser. EPTCS, vol. 325, pp. 164–177 (2020)

10. Guerrero, C., Lera, I., Juiz, C.: Migration-aware genetic optimization for mapreduce scheduling and replica placement in hadoop. J. Grid Comput. **16**(2), 265–284 (2018)

11. Fernando, N., Loke, S.W., Avazpour, I., Chen, F., Abkenar, A.B., Ibrahim, A.: Opportunistic fog for IoT: challenges and opportunities. IEEE Internet Things J. **6**(5), 8897–8910 (2019)

12. Casadei, R., Fortino, G., Pianini, D., Russo, W., Savaglio, C., Viroli, M.: Modelling and simulation of opportunistic IoT services with aggregate computing. Future Gener. Comput. Syst. **91**, 252–262 (2019)

13. Lera, I., Guerrero, C., Juiz, C.: Algoritmo descentralizado para la asignación de servicios en arquitecturas de fog computing basado en un proceso expansivo de migración de instancias in Jornadas SARTECO (2019)

14. Dazzi, P., Mordacchini, M.: Scalable decentralized indexing and querying of multi-streams in the fog. J. Grid Comput. **18**(3), 395–418 (2020). https://doi.org/10.1007/s10723-020-09521-3

15. Brogi, A., Forti, S., Guerrero, C., Lera, I.: Towards declarative decentralised application management in the fog. In: GAUSS (2020). In press

16. Lera, I., Guerrero, C., Juiz, C.: YAFS: A simulator for IoT scenarios in Fog computing. IEEE Access **7**, 91745–91758 (2019)

17. Bracciale, L., Bonola, M., Loreti, P., Bianchi, G., Amici, R., Rabuffi, A.: CRAWDAD dataset roma/taxi (v. 2014-07-17). https://crawdad.org/roma/taxi/20140717/taxicabs (2014)

18. Urry, L.A., Cain, M.L., Wasserman, S., Minorsky, P., Jane, R. Campbell Biology, 11th edn. Pearson, London (2017)

19. Bayles, K.W.: Bacterial programmed cell death: making sense of a paradox. Nature Rev. Microbiol. **12**(1), 63–69 (2014)

20. Brogi, A., Forti, S., Gaglianese, M.: Measuring the fog, gently. In: ICSOC, pp. 523–538 (2019)

21. Forti, S., Gaglianese, M., Brogi, A.: Lightweight self-organising distributed monitoring of Fog infrastructures. Future Gener. Comput. Syst. **114**, 605–618 (2021). (In press)

22. Taherizadeh, S., Jones, A.C., Taylor, I., Zhao, Z., Stankovski, V.: Monitoring self-adaptive applications within edge computing frameworks: a state-of-the-art review. J. Syst. Softw. **136**, 19–38 (2018)

23. Guerrero, C., Lera, I., Juiz, C.: Evaluation and efficiency comparison of evolutionary algorithms for service placement optimization in fog architectures. Future Gener. Comput. Syst. **97**, 131–144 (2019)

24. Drabent, W.: The prolog debugger and declarative programming. In: International Symposium on Logic-Based Program Synthesis and Transformation, pp. 193–208. Springer (2019)

25. Brogi, A., Forti, S., Guerrero, C., Lera, I.: Meet genetic algorithms in monte carlo: optimised placement of multi-service applications in the fog. In: 2019 IEEE International Conference on Edge Computing (EDGE). IEEE, pp. 13–17 (2019)

26. Pietri, I., Sakellariou, R.: Mapping virtual machines onto physical machines in cloud computing: a survey. ACM Comput. Surv. **49**(3), 1–30 (2016)

27. Tomarchio, O., Calcaterra, D., Modica, G.D.: Cloud resource orchestration in the multi-cloud landscape: a systematic review of existing frameworks. J. Cloud Comput. **9**, 49 (2020). https://doi.org/10.1186/s13677-020-00194-7

28. Kadioglu, S., Colena, M., Sebbah, S.: Heterogeneous resource allocation in Cloud Management. In: NCA, pp. 35–38 (2016)

29. Yin, Q., Schüpbach, A., Cappos, J., Baumann, A., Roscoe, T.: Rhizoma: a runtime for self-deploying, self-managing overlays. In: Middleware 2009, pp. 184–204 (2009)

30. Carlini, E., Coppola, M., Dazzi, P., Mordacchini, M.: A Passarella, Self-optimising decentralised service placement in heterogeneous cloud federation. In: SASO, pp. 110–119 (2016)

31. Sathiaseelan, A., Selimi, M., Molina, C., Lertsinsrubtavee, A., Navarro, L., Freitag, F., Ramos, F., Baig, R.:

Towards decentralised resilient community clouds. In: MECC, pp. 1–6 (2017)

32. Ferrer, A.J., Marquès, J.M., Jorba, J.: Towards the decentralised cloud: survey on approaches and challenges for mobile, ad hoc, and edge computing. ACM Comput. Surv. **51**(6), 1–36 (2019)

33. Xiang, Z., Deng, S., Taheri, J., Zomaya, A.Y.: Dynamical service deployment and replacement in resource-constrained edges. Mob. Netw. Appl. **25**(2), 674–689 (2020). https://doi.org/10.1007/s11036-019-01449-7

34. Colistra, G., Pilloni, V., Atzori, L.: The problem of task allocation in the internet of things and the consensus-based approach. Comput. Networks **73**, 98–111 (2014)

35. Mennes, R., Spinnewyn, B., Latré, S., F. Botero, J.: GRECO: A distributed genetic algorithm for reliable application placement in hybrid clouds. In: CloudNet, pp. 14–20 (2016)

36. Guerrero, C., Lera, I., Juiz, C.: A lightweight decentralized service placement policy for performance optimization in fog computing. J. Ambient Intell. Humaniz. Comput. **10**(6), 2435–2452 (2019)

37. Herrera, J., Moltó, G.: Toward bio-inspired auto-scaling algorithms: An elasticity approach for container orchestration platforms. IEEE Access **8**, 52139–52150 (2020)

38. Rossi, F., Cardellini, V., Lo Presti, F., Nardelli, M.: Geo-distributed efficient deployment of containers with kubernetes. Comput. Commun. **159**, 161–174 (2020)

39. Hinrichs, T.L., Gude, N.S., Casado, M., Mitchell, J.C., Shenker, S.: Practical declarative network management. In: WREN, pp. 1–10 (2009)

40. Herden, S., Zwanziger, A., Robinson, P.: Declarative application deployment and change management. In: CNSM, pp. 126–133 (2010)

41. Forti, S., Paganelli, F., Brogi, A.: Probabilistic QoS-aware placement of VNF chains at the Edge. Theory Pract. Log. Program. In press (2021)

42. Forti, S., Ferrari, G., Brogi, A.: Secure Cloud-Edge deployments, with trust. Future Gener. Comput. Syst. **102**, 775–788 (2020)

43. Harzenetter, L., Breitenbücher, U., Leymann, F., Saatkamp, K., Weder, B., Wurster, M.: Automated generation of management workflows for applications based on deployment models. In: EDOC, pp. 216–225 (2019)

44. Casadei, R., Viroli, M.: Coordinating computation at the edge: a decentralized, self-organizing, spatial approach. In:

FMEC 2019, pp. 60–67 (2019). https://doi.org/10.1109/FMEC.2019.8795355

45. Pianini, D., Casadei, R., Viroli, M., Natali, A.: Partitioned integration and coordination via the self-organising coordination regions pattern. Future Gener. Comput. Syst. **114**, 44–68 (2021)

46. Viroli, M., Beal, J., Damiani, F., Audrito, G., Casadei, R., Pianini, D.: From field-based coordination to aggregate computing. In: COORDINATION 2018, vol. 10852, pp. 252–279. Springer (2018)

47. Margariti, S.V., Dimakopoulos, V.V., Tsoumanis, G.: Modeling and simulation tools for fog computing–a comprehensive survey from a cost perspective. Future Internet **12**(5), 89 (2020)

48. Gupta, H., Vahid Dastjerdi, A., Ghosh, S.K., Buyya, R.: iFogSim: A toolkit for modeling and simulation of resource management techniques in the internet of things, edge and fog computing environments. Soft. Pract. Exp. **47**(9), 1275–1296 (2017)

49. Sonmez, C., Ozgovde, A., Ersoy, C.: EdgeCloudSim: An environment for performance evaluation of edge computing systems. Trans. Emerg. Telecommun. Technol. **29**, e3493 (2018)

50. Forti, S., Pagiaro, A., Brogi, A.: Simulating FogDirector application management. Simul. Model. Pract. Theory **101**(102021), 1–18 (2020)

51. Kurdi, H.A., Aldaood, M.F., Al-Megren, S., Aloboud, E., Aldawood, A.S., Youcef-Toumi, K.: Adaptive task allocation for multi-uav systems based on bacteria foraging behaviour. Appl. Soft Comput. **83** (2019)

52. Ahsan, M.M., Gupta, K.D., Nag, A.K., Pouydal, S., Kouzani, A.Z., Mahmud, M.P.: Applications and evaluations of bio-inspired approaches in cloud security: a review. IEEE Access (2020)

53. Chang, Y.-C., Cai, W.-X., Jhuang, J.-W.: Bacteria-inspired communication mechanism based on software-defined network. In: WOCC, pp. 1–3 (2018)

54. Chao, H., Cho, H., Shih, T.K., Chen, C.: Bacteria-inspired network for 5g mobile communication. IEEE Netw. **33**(4), 138–145 (2019)