Continuous Reasoning for Managing Next-Gen Distributed Applications

Stefano Forti and Antonio Brogi
Department of Computer Science, University of Pisa, Italy
{stefano.forti, antonio.brogi}@di.unipi.it

Continuous reasoning has proven effective in incrementally analysing changes in application codebases within Continuous Integration/Continuous Deployment (CI/CD) software release pipelines. In this article, we present a novel declarative continuous reasoning approach to support the management of multi-service applications over the Cloud-IoT continuum, in particular when infrastructure variations impede meeting application's hardware, software, IoT or network QoS requirements. We show how such an approach brings considerable speed-ups compared to non-incremental reasoning.

1 Introduction

Large IT companies rely on *continuous reasoning* to support iterative software development up to its *continuous integration* within a single shared codebase [18]. For instance, static analyses based on compositional models of programs and code repositories permit exploiting continuous reasoning to check security properties on a whole codebase repository by verifying them only on the changes occurred since the last performed verification. By employing such techniques, Facebook Infer incrementally and continuously checks whether new commits from Facebook developers are safe (and can be accepted in the codebase) or not (and should be revised before acceptance) [9]. This dramatically reduces analysis times, instead of repeating a complete codebase analysis at every single commit.

Meanwhile, the Internet of Things (IoT) is undergoing a relentless growth which is becoming more difficult to support with the current software and infrastructure architectures. Besides, many next-gen IoT applications will have QoS requirements such as low latencies, network bandwidth availability and deployment security, which are difficult to guarantee over Cloud-only infrastructures [4]. Relying upon a large-scale hierarchy of distributed nodes, Cloud-IoT computing paradigms aim at enabling the QoS- and context-aware deployment of next-gen IoT application services to any device supporting them over the Cloud-IoT computing continuum, e.g. in Fog [1] or Osmotic [22] computing. In this scenario, the problem of deciding where to place application services (i.e., functionalities) to infrastructure nodes is of primary importance and it is provably NP-hard [2, 7]. In recent years, much literature focussed on determining the best QoS- and context-aware placement of multi-service IoT applications to Cloud-IoT infrastructures, mainly exploiting search-based and mathematical programming solutions [4].

Inspired from continuous reasoning and from our previous work on next-gen application placement [5] and management [11], we consider it promising to extend the concept of continuous reasoning in support of the management of next-gen multi-service distributed applications. This will reduce the time needed to make management decisions when only part of a running application deployment is affected by changes in the Cloud-IoT infrastructures, e.g. crash of a node hosting a service, degraded network QoS in between communicating services running on different nodes. By mainly considering the migration of services suffering due to such infrastructure changes, continuous reasoning can bring

F. Ricca, A. Russo et al. (Eds.): Proc. 36th International Conference on Logic Programming (Technical Communications) 2020 (ICLP 2020)
EPTCS 325, 2020, pp. 164–177, doi:10.4204/EPTCS.325.22

two-fold benefits. On one hand, it permits scaling to larger sizes of the placement problem by incrementally solving smaller instances of such a problem, thus reducing the time needed to make informed decisions. On the other hand, it can reduce the number of management operations needed to adapt the current deployment to the new infrastructure conditions, by avoiding unnecessary service migrations.

In this article, we move a first step towards this direction by introducing the application of continuous reasoning in support of the automated management of next-gen multi-service applications in Cloud-IoT settings. We propose a novel declarative methodology and its open-source continuous reasoner prototype, FogBrain, that makes informed service migration decisions, when infrastructure variations prevent running applications from meeting their hardware, software, IoT or network QoS requirements. Our methodology shows three main elements of novelty, corresponding to very desirable properties for Cloud-IoT application management support:

- it is *declarative*, hence more concise, easier to understand, modify and maintain when compared to existing procedural solutions, and it is also characterised by a high level of flexibility and extensibility, which suits the ever-changing needs of Cloud-IoT scenarios,
- it is intrinsically *explainable* as it derives proofs for input user queries by relying on Prolog state-of-the-art resolution engines, and it can be easily extended to justify *why* a certain management decision was taken at runtime in the spirit of explainable AI (XAI), and
- it features *scalability* by supporting application management at the large-scale, by relying on a continuous reasoning approach to reduce the size of the considered problem instance only to those application services currently in need for attention.

The rest of this paper is organised as follows. We first illustrate how FogBrain determines context- and QoS-aware placements of multiservice applications to Cloud-IoT infrastructures (Sect. 2), and how it implements continuous reasoning for runtime placement decisions (Sect. 3). We then discuss the scalability of FogBrain over examples of increasing size (Sect. 4), and survey some related work (Sect. 5). We finally conclude and highlight directions for future work (Sect. 6).

2 Placement of Next-Gen Distributed Applications

Hereafter, we describe a declarative solution to the problem of placing multi-service applications to Cloud-IoT computing infrastructures in a context- and QoS-aware manner. Eligible placements are mappings of each service of an application to a computational node in the available infrastructure, which supports all service software, hardware, IoT and QoS requirements. Placements can include nodes that host more than one service, if their capabilities are sufficient. The Prolog program described in this section will be the core of the FogBrain prototype that we will present in Sect. 3.

Declaring Application Requirements. First, applications identified by `AppId` and made from services `ServiceId1, ..., ServiceIdK` are declared as in

```
app(AppId, [ServiceId1, ..., ServiceIdK]).
```

Second, application services identified by `ServiceId` and associated with their software `SwReqs`, hardware `HwReqs` and IoT requirements `TReqs` are declared as in

```
service(ServiceId, SwReqs, HwReqs, TReqs).
```

Finally, interactions between services `ServiceId1` and `ServiceId2`, associated with their maximum latency `LatReq` and minimum bandwidth `BwReq` requirements are declared as in

```
s2s(ServiceId1, ServiceId2, LatReq, BwReq).
```

Example. Consider the Virtual Reality (VR) application in Fig. 1, which is made of three interacting services. VR videos are streamed from the Video Storage service to the VR Driver service, connected to a VR Viewer device, passing through the Scene Selector service, which selects the portion of the video to show to the user, based on her currently sensed head positioning.

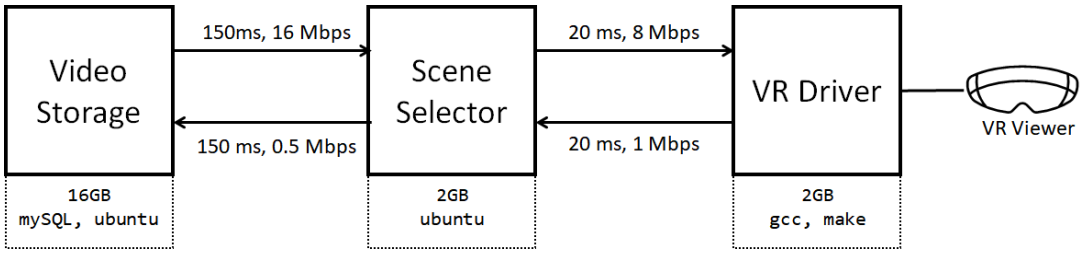


Figure 1: VR application of the example.

The requirements of such an application can be easily declared[1] as in Fig. 2.

```
application(vrApp, [videoStorage, sceneSelector, vrDriver]).
service(videoStorage, [mySQL, ubuntu], 16, []).
service(sceneSelector, [ubuntu], 2, []).
service(vrDriver, [gcc, make], 2, [vrViewer]).
s2s(videoStorage, sceneSelector, 150, 16).
s2s(sceneSelector, videoStorage, 150, 0.5).
s2s(sceneSelector, vrDriver, 20, 8).
s2s(vrDriver, sceneSelector, 20, 1).
```

Figure 2: Example VR application specification.

For instance, the VR Driver service requires both `gcc` and `make` to bootstrap on the target deployment node, the availability of 2 hardware units and the reachability of the specified vrViewer. Analogously, the service-to-service link between the Scene Selector and the VR Driver requires end-to-end download bandwidth of at least 8 Mbps with at most 20 ms latency to stream videos to the VR Viewer, and end-to-end upload bandwidth of at least 1 Mbps with at most 20 ms latency to receive the currently sensed user's head positioning (as per the last two `s2s/4` facts in Fig. 2). ◇

Declaring Infrastructure Capabilities. Dually to application requirements, it is possible to declare an infrastructure node identified by `NodeId` and its software (i.e. `SwCaps`), hardware (i.e. `HwCaps`) and IoT (i.e. `TCaps`) capabilities, obtained via infrastructure monitoring, as in

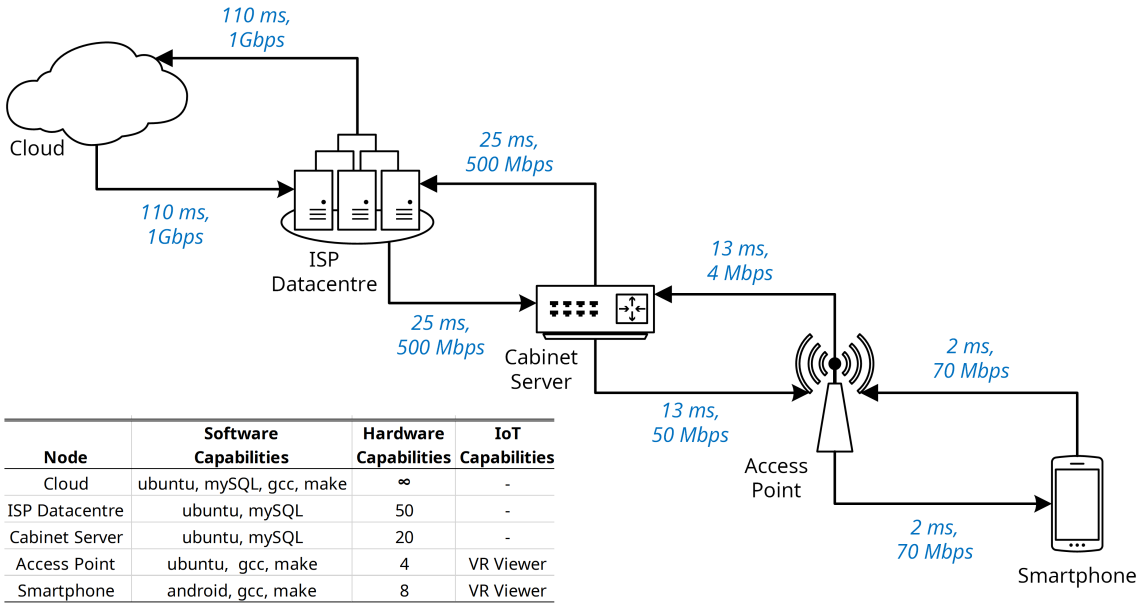
```
node(NodeId, SwCaps, HwCaps, TCaps).
```

Besides, it is possible to declare the monitored end-to-end latency (i.e. `FeatLat`) and bandwidth (i.e. `FeatBw`) featured by communication links between a pair of nodes `NodeIdA`–`NodeIdB` as in

```
link(NodeIdA, NodeIdB, FeatLat, FeatBw).
```

[1] For the sake of simplicity, we consider generic hardware units to express the hardware requirements to be met, as in most of the approaches we surveyed in [4].

Example. Consider the infrastructure of Fig. 3, sketching a Cloud-IoT computing continuum that spans through the Internet backbone, an ISP infrastructure, a metropolitan access network, and eventually reaches a home access point enabling a wireless LAN. Assume that both the access point and the smartphone can reach out a VR Viewer in the wireless LAN enabled by the access point. Such VR Viewer can be exploited by a running instance of the application of Fig. 1.



Node	Software Capabilities	Hardware Capabilities	IoT Capabilities
Cloud	ubuntu, mySQL, gcc, make	∞	-
ISP Datacentre	ubuntu, mySQL	50	-
Cabinet Server	ubuntu, mySQL	20	-
Access Point	ubuntu, gcc, make	4	VR Viewer
Smartphone	android, gcc, make	8	VR Viewer

Figure 3: Infrastructure of the example.

Such an infrastructure can be represented, including end-to-end links not shown in Fig. 3, as listed in Fig. 4.

```
node(cloud, [ubuntu, mySQL, gcc, make], inf, []).
node(ispdatacentre, [ubuntu, mySQL], 50, []).
node(cabinetserver, [ubuntu, mySQL], 20, []).
node(accesspoint, [ubuntu, gcc, make], 4, [vrViewer]).
node(smartphone, [android, gcc, make], 8, [vrViewer]).

link(cloud, ispdatacentre, 110, 1000).          link(cabinetserver, accesspoint, 13, 50).
link(cloud, cabinetserver, 135, 100).           link(cabinetserver, smartphone, 15, 35).
link(cloud, accesspoint, 148, 20).              link(accesspoint, cloud, 148, 3).
link(cloud, smartphone, 150, 18 ).              link(accesspoint, ispdatacentre, 38, 4).
link(ispdatacentre, cloud, 110, 1000).          link(accesspoint, cabinetserver, 13, 4).
link(ispdatacentre, cabinetserver, 25, 500).    link(accesspoint, smartphone, 2, 70).
link(ispdatacentre, accesspoint, 38, 50).       link(smartphone, cloud, 150, 2).
link(ispdatacentre, smartphone, 40, 35).        link(smartphone, ispdatacentre, 40, 2.5).
link(cabinetserver, cloud, 135, 100).           link(smartphone, cabinetserver, 15, 3).
link(cabinetserver, ispdatacentre, 25, 500).    link(smartphone, accesspoint, 2, 70).
```

Figure 4: Example infrastructure declaration.

The proposed representation captures asymmetric links (i.e. with different upload and download bandwidth), which are common in Cloud-IoT scenarios, especially in edge networks. ◇

Determining Eligibile Application Placements. It is now possible to define a declarative *generate & test* strategy to match application requirements to infrastructure capabilities and determine eligible placements as per the code of Fig. 5. The `placement/2` predicate (lines 1–4) retrieves an application `App` (line 2) and looks for a `Placement` of its `Services` that meets all the application requirements over the available infrastructure by calling the predicate `placement/7` (line 3). A found eligible placement is asserted as a fact into the knowledge base of the FogBrain reasoner along with the associated hardware `AllocHW` and bandwidth `AllocBW` allocations (line 4).

The `placement/7` predicate (lines 5–9) recursively scans the list of services to be deployed (i.e. `[S|Ss]`) to determine an eligible `Placement` for each of them. The variables `AllocHW`/`NewAllocHW` and `AllocBW`/`NewAllocBw` are used to keep track of the hardware and bandwidth allocated by the placement being built in `Placement`/`NewPlacement` (line 6). This information is used not to exceed the hardware capabilities nor to saturate communication links with cumulative hardware or bandwidth requirements of application services mapped onto a same node or link, respectively. When a valid placement for service `S` on node `N` is found, it is appended to the placement being built as the tuple `on(S,N)` (line 9). To this end, `placement/7` coordinates the checks on service requirements (line 7) with the checks on service-to-service communication requirements (line 8) in two repeated steps, until the list of services to be deployed is empty (line 5).

As a first step, `servicePlacement/4` (lines 7, 10–16) non-deterministically places service `S` onto a node `N` (lines 7, 10–16), and checks that node `N` can at least support the hardware requirements of `S` (line 13) and meets its IoT (lines 14, 17) and software (lines 15, 18) requirements. Then, by means of `hwReqsOK/5` (lines 16, 19–24), `servicePlacement/4` performs a check on cumulative hardware allocation, as it updates the `AllocHW` accumulator into the `NewAllocHW` accumulator by summing up the requirements of `S` to the hardware previously allocated to other services mapped onto `N` as per `Placement`. Thus, the hardware `AllocHW` currently allocated at any node `N` by the current placement can always be found within the list `TAllocHW` (line 7) in the couple `(N, AllocHW)`.

As a second step, `flowOK/5` (lines 8, 25–27) checks whether service-to-service interactions between the last-placed service `S` and previously placed services in `Placement` can support the required network QoS. All the requirements of latency and bandwidth to be checked on the communication link between the nodes `N` and `N2` (or `N1` and `N`) that support communicating services `S` and `S2` (or `S1` and `S`), respectively, are retrieved by exploiting the `interested/8` predicate (lines 26, 28–31). Then, the `serviceFlowOK/3` (line 27, 32–37) recursively scans the `N2Ns` list, and it checks latency requirements (line 35) and cumulative bandwidth allocation (via `bwOK/6`, line 36, 38–43), and updates the `AllocBw` accumulator accordingly. Overall, this second step implements a *fail-fast* heuristic, by immediately backtracking from a placement that cannot meet requirements on communication QoS with other services.

Finally, it is worth noting that `hwReqsOK/5` and `bwOK/6` rely upon two user-set threshold facts

```
hwTh(THW). % 0.5 by default
bwTh(TBW). % 0.2 by default
```

which represent the amount of hardware and bandwidth not to be allocated so to avoid overloading nodes and links, respectively.

Example. By querying the `placement/2` predicate on the example application (Fig. 1) and infrastructure (Fig. 3), it outputs

```
1 ?- placement(vrApp, P).
P = [on(vrDriver, accesspoint), on(sceneSelector, cabinetserver), on(videoStorage, cloud)]
```

```
placement(App, Placement) :-
  application(App, Services),
  placement(Services, [], AllocHW, [], AllocBW, [], Placement),
  assert(deployment(App, Placement, AllocHW, AllocBW)).

placement([], AllocHW, AllocHW, AllocBW, AllocBW, Placement, Placement).
placement([S|Ss], AllocHW, NewAllocHW, AllocBW, NewAllocBW, Placement, NewPlacement) :-
  servicePlacement(S, AllocHW, TAllocHW, N),
  flowOK(S, N, Placement, AllocBW, TAllocBW),
  placement(Ss, TAllocHW, NewAllocHW, TAllocBW, NewAllocBW, [on(S,N)|Placement], NewPlacement).

servicePlacement(S, AllocHW, NewAllocHW, N) :-
  service(S, SWReqs, HWReqs, TReqs),
  node(N, SWCaps, HWCaps, TCaps),
  hwTh(T), HWCaps >= HWReqs + T,
  thingReqsOK(TReqs, TCaps),
  swReqsOK(SWReqs, SWCaps),
  hwReqsOK(HWReqs, HWCaps, N, AllocHW, NewAllocHW).

thingReqsOK(TReqs, TCaps) :- subset(TReqs, TCaps).

swReqsOK(SWReqs, SWCaps) :- subset(SWReqs, SWCaps).

hwReqsOK(HWReqs, HWCaps, N, [], [(N,HWReqs)]) :-
  hwTh(T), HWCaps >= HWReqs + T.
hwReqsOK(HWReqs, HWCaps, N, [(N,AllocHW)|L], [(N,NewAllocHW)|L]) :-
  NewAllocHW is AllocHW + HWReqs, hwTh(T), HWCaps >= NewAllocHW + T.
hwReqsOK(HWReqs, HWCaps, N, [(N1,AllocHW)|L], [(N1,AllocHW)|NewL]) :-
  N \== N1, hwReqsOK(HWReqs, HWCaps, N, L, NewL).

flowOK(S, N, Placement, AllocBW, NewAllocBW) :-
  findall(n2n(N1,N2,ReqLat,ReqBW), interested(N1,N2,ReqLat,ReqBW,S,N,Placement), Ss),
  serviceFlowOK(Ss, AllocBW, NewAllocBW).

interested(N, N2, ReqLat, ReqBW, S, N, Placement) :-
  s2s(S, S2, ReqLat, ReqBW), member(on(S2,N2), Placement), N\==N2.
interested(N1, N, ReqLat, ReqBW, S, N, Placement) :-
  s2s(S1, S, ReqLat, ReqBW), member(on(S1,N1), Placement), N\==N1.

serviceFlowOK([], AllocBW, AllocBW).
serviceFlowOK([n2n(N1,N2,ReqLat,ReqBW)|Ss], AllocBW, NewAllocBW) :-
  link(N1, N2, FeatLat, FeatBW),
  FeatLat =< ReqLat,
  bwOK(N1, N2, ReqBW, FeatBW, AllocBW, TAllocBW),
  serviceFlowOK(Ss, TAllocBW, NewAllocBW).

bwOK(N1, N2, ReqBW, FeatBW, [], [(N1,N2,ReqBW)]):-
  bwTh(T), FeatBW >= ReqBW + T.
bwOK(N1, N2, ReqBW, FeatBW, [(N1,N2,AllocBW)|L], [(N1,N2,NewAllocBW)|L]):-
  NewAllocBW is ReqBW + AllocBW, bwTh(T), FeatBW >= NewAllocBW + T.
bwOK(N1, N2, ReqBW, FeatBW, [(N3,N4,AllocBW)|L], [(N3,N4,AllocBW)|NewL]):-
  \+ (N1 == N3, N2 == N4), bwOK(N1,N2,ReqBW,FeatBW,L,NewL).
```

Figure 5: Multi-service application placement to infrastructure nodes.

The obtained placement is one out of the $5^3 = 125$ that are combinatorially possible, and one of the 12 eligible ones. DevOps engineers managing the VR application can select this placement to initially deploy it to the available infrastructure. ◇

We conclude this section with a note on the time complexity of the proposed solution. Having to place S services onto N nodes, our prototype could explore at most the whole search space to determine an eligible placement (if any), incurring in a worst-case time complexity of $O(N^S)$.

3 Continuous Management of Next-Gen Distributed Applications

In this section, we illustrate how FogBrain implements continuous reasoning on application placement decisions. We only describe the main predicates of FogBrain realising such a feature. The complete FogBrain codebase and interactive documentation, realised with Klipse [20], is available online[2].

FogBrain works as per the high-level behaviour of `fogBrain/2` shown in Fig. 6. We assume that FogBrain is embedded in an infinite loop and that changes in the infrastructure conditions trigger a query to `fogBrain(App, Placement)`, which distinguishes the case of runtime application management (lines 44–46) from the case of first application deployment (lines 47–49). In the former case, since a `deployment/4` has already been asserted for `App` (line 45), FogBrain triggers a continuous reasoning step on the existing deployment so to understand which management actions are possibly needed, if the deployment does not currently meet all application requirements (line 46). In the latter case, where no previous deployment of `App` exists (line 48), FogBrain queries the `placement/2` predicate of Fig. 5 (line 49) to determine a first placement for the input application.

```
44 fogBrain(App, NewPlacement) :-
45     deployment(App, Placement, AllocHW, AllocBW),
46     reasoningStep(App, Placement, AllocHW, AllocBW, NewPlacement).
47 fogBrain(App, Placement) :-
48     \+ deployment(App,_,_,_),
49     placement(App, Placement).
```

Figure 6: Overview of FogBrain.

The key idea of FogBrain is to reduce the time needed to make informed application management decisions by limiting the size of the considered placement problem instance, so to tame its NP-hard nature and worst-case exp-time complexity. This is achieved by building suitable partially ground queries for `placement/7` via the `reasoningStep/5` predicate, which sets to unbound the placement of those application services that, due to the last infrastructure changes, cannot meet their hardware, software or QoS requirements anymore. Then, a new placement will be determined only for those application services suffering from the current node or network conditions. As a consequence, a problem instance to be solved is handled much faster in most of the cases. Only when it will not be possible to determine an eligible migration for the suffering services, FogBrain will look for a new complete placement.

We now detail how `reasoningStep/5`, shown in Fig. 7, works. The `reasoningStep/5` predicate (lines 50–52) determines via `toMigrate/2` (lines 51, 53–56) which `ServicesToMigrate` in the current `Placement` need to be migrated due to node or network conditions that impede meeting application requirements. Then, it exploits the `replacement/7` (line 52) to query the `placement/7` with partially ground placement information, so to attempt migrating only suffering services.

[2] FogBrain is available at `https://pages.di.unipi.it/forti/fogbrain`

```
50 reasoningStep(App, Placement, AllocHW, AllocBW, NewPlacement) :-
51     toMigrate(Placement, ServicesToMigrate),
52     replacement(App, ServicesToMigrate, Placement, AllocHW, AllocBW, NewPlacement).

53 toMigrate(Placement, ServicesToMigrate) :-
54     findall((S,N,HWReqs), onSufferingNode(S,N,HWReqs,Placement), ServiceDescr1),
55     findall((SD1,SD2), onSufferingLink(SD1,SD2,Placement), ServiceDescr2),
56     merge(ServiceDescr2, ServiceDescr1, ServicesToMigrate).

57 onSufferingNode(S, N, HWReqs, Placement) :-
58     member(on(S,N), Placement),
59     service(S, SWReqs, HWReqs, TReqs),
60     nodeProblem(N, SWReqs, TReqs).

61 onSufferingLink((S1,N1,HWReqs1),(S2,N2,HWReqs2),Placement) :-
62     member(on(S1,N1), Placement), member(on(S2,N2), Placement), N1 \== N2,
63     s2s(S1, S2, ReqLat, _),
64     communicationProblem(N1, N2, ReqLat),
65     service(S1, _, HWReqs1, _),
66     service(S2, _, HWReqs2, _).
```

Figure 7: The `reasoningStep/5` predicate.

The `toMigrate/2` predicate (lines 53–56) retrieves all service descriptors of those services that are currently suffering due to node or communication issues via predicates `onSufferingNode/4` (line 54) and `onSufferingLink/3` (line 55), respectively. Service descriptors are triples `(S,N,HWReqs)` containing the service identifier, the current deployment node and the service hardware requirements.

First, the `toMigrate/2` predicate calls `onSufferingNode/4` (lines 57–60) to determine all services suffering due to node changes. In turn, `onSufferingNode/4` exploits `nodeProblem/3` (line 60) to check whether the node onto which a service `S` is deployed as per the current `Placement` (lines 58–59) cannot satisfy all requirements of `S`. Analogously, to determine all pairs of service descriptors `(SD1, SD2)` suffering from communication problems, `toMigrate/2` calls `onSufferingLink/3` (lines 61-66). In turn, `onSufferingLink/3` exploits `communicationProblem/3` (line 69) to check if the link supporting the communication between `S1` and `S2` (lines 62–63) does not feature suitable QoS.

Both `nodeProblem/3` and `communicationProblem/3` can be flexibly defined by FogBrain users to check different properties. Fig. 8 shows the two default definitions of the predicates checking node and links current capabilities against software, IoT and communication QoS requirements of the application services as well as nodes and links overloading situations (lines 67–69 and 72–73). The default definitions also check nodes and links for failures (lines 70–71 and 75–76, respectively).

The `merge/3` predicate (line 56) merges into the list `ServicesToMigrate` all found descriptors of suffering services, by removing duplicates and splitting descriptor couples determined by `onSufferingLink/3`. Such a list, along with the current `Placement`, hardware and bandwidth allocations, is passed to predicate `replacement/6` (line 52), with the goal of determining a `NewPlacement` for those suffering services.

We now describe the functioning of `replacement/6`, listed in Fig. 9. In case no service needs to be migrated, the `NewPlacement` does not change, i.e. it coincides with the current `Placement` (line 77). Otherwise, the current `deployment` is retracted, and FogBrain builds a suitable partially-ground query to `placement/7` (line 84) by means of the predicates `partialPlacement/3` (lines 80–81), `freeHWAllocation/3` (lines 82) and `freeBWAllocation/4` (lines 83).

```
67 nodeProblem(N, SWReqs, TReqs) :-
68     node(N, SWCaps, HWCaps, TCaps),
69     hwTh(T), \+ (HWCaps > T, thingReqsOK(TReqs,TCaps), swReqsOK(SWReqs,SWCaps)).
70 nodeProblem(N, _, _) :-
71     \+ node(N, _, _, _).

72 communicationProblem(N1, N2, ReqLat) :-
73     link(N1, N2, FeatLat, FeatBW),
74     (FeatLat > ReqLat; bwTh(T), FeatBW < T).
75 communicationProblem(N1,N2,_) :-
76     \+ link(N1, N2, _, _).
```

Figure 8: Default `nodeProblem/3` and `communicationProblem/3` predicates.

```
77 replacement(_, [], Placement, _, _, Placement).
78 replacement(A, ServicesToMigrate, Placement, AllocHW, AllocBW, NewPlacement) :-
79     ServicesToMigrate \== [], retract(deployment(A, Placement, _, _)),
80     findall(S, member((S,_,_), ServicesToMigrate), Services),
81     partialPlacement(Placement, Services, PPlacement),
82     freeHWAllocation(AllocHW, PAllocHW, ServicesToMigrate),
83     freeBWAllocation(AllocBW, PAllocBW, ServicesToMigrate, Placement),
84     placement(Services, PAllocHW, NewAllocHW, PAllocBW, NewAllocBW, PPlacement, NewPlacement),
85     assert(deployment(A, NewPlacement, NewAllocHW, NewAllocBW)).

86 partialPlacement([],_,[]).
87 partialPlacement([on(S,_)|P],Services,PPlacement) :-
88     member(S,Services), partialPlacement(P,Services,PPlacement).
89 partialPlacement([on(S,N)|P],Services,[on(S,N)|PPlacement]) :-
90     \+member(S,Services), partialPlacement(P,Services,PPlacement).
```

Figure 9: The `replacement/7` and `partialPlacement/3` predicates.

First of all, `partialPlacement/3` (lines 86–90) recursively scans the current `Placement` and removes from it all services to be migrated so to build the partial placement `PPlacement`. Then, the predicate `freeHWAllocation/3`, listed in Fig. 10, cleans up the current hardware allocation by removing the hardware allocation of suffering services from the nodes whose services are migrated. This is done by recursively scanning the current allocation for each node `(N, AllocHW)`, summing all hardware requirements of the descriptors of the services to be migrated from `N` via `sumNodeHWToFree` (line 93, 97–99), and by removing them from `AllocHW` (line 94–95). All updated hardware allocations are assembled together into `NewAllocHW` via `assemble/3` (lines 100–101), while removing zero-hardware allocations (line 100). Analogously, the `freeBWAllocation/4` cleans up the current bandwidth allocation `BWAlloc` by removing from communication links all bandwidth allocations that involve at least one suffering service.

Finally, as listed in Fig. 9, the partial placement `PPlacement`, the partial hardware and bandwidth allocations, `PAllocHW` and `PAllocBW`, and the identifiers of the `Services` to be migrated are input to `placement/7` to determine an eligible `NewPlacement` (and the related `NewAllocHW` and `NewAllocBW`) (line 84) to be asserted as a `deployment/4` for the application `A` (line 85). If partially migrating application services fails, no `deployment/4` is asserted and a complete new placement is looked for as per the second clause of `fogBrain/2` (line 47–49, Fig. 6).

```
freeHWAllocation([], [], _).
freeHWAllocation([(N,AllocHW)|L], NewL, ServicesToMigrate) :-
    sumNodeHWToFree(N, ServicesToMigrate, HWToFree),
    NewAllocHW is AllocHW - HWToFree,
    freeHWAllocation(L, TempL, ServicesToMigrate),
    assemble((N,NewAllocHW), TempL, NewL).

sumNodeHWToFree(_, [], 0).
sumNodeHWToFree(N, [(_,N,H)|STMs], Tot) :- sumNodeHWToFree(N, STMs, HH), Tot is H+HH.
sumNodeHWToFree(N, [(_,N1,_)|STMs], H) :- N \== N1, sumNodeHWToFree(N, STMs, H).

assemble((_,NewAllocHW), L, L) :- NewAllocHW=:=0.
assemble((N, NewAllocHW), L, [(N,NewAllocHW)|L]) :- NewAllocHW>0.
```

Figure 10: The `freeHWAllocation/3` predicate.

Example. Retaking the example of Sect. 2, we now exploit FogBrain to determine a first eligible deployment for the VR application:

```
1 ?- fogBrain(vrApp,P).
P = [on(vrDriver, accesspoint), on(sceneSelector, cabinetserver), on(videoStorage, cloud)] .
```

Assuming now that the Cloud datacentre does not offer anymore the software capabilities required by Video Storage (i.e. `node(cloud, [centos, gcc, make], inf, []).`), we obtain:

```
2 ?- fogBrain(vrApp,P).
P = [on(videoStorage, ispdatacentre), on(vrDriver, accesspoint),
     on(sceneSelector, cabinetserver)]
```

This reasoning step suggests migrating the Video Storage from the Cloud to the ISP datacentre. Such a result was obtained by exploiting the following partially ground query of `placement/7`:

```
placement([videoStorage], [(cabinetserver,2),(accesspoint,2)], NewAllocHW,
    [(accesspoint,cabinetserver,1),(cabinetserver,accesspoint,8)], NewAllocBW,
    [on(vrDriver,accesspoint),on(sceneSelector,cabinetserver)], NewPlacement).
```

obtained as per the `reasoningStep/5` predicate, by removing Video Storage from the input partial placement, and from the input hardware and bandwidth allocations, while leaving untouched information related to non-suffering services (viz. VR Driver and Scene Selector). ◇

We conclude this section with a note on the time complexity of our continuous reasoning solution. Identifying suffering services and building the partial query for `placement/7` incurs in worst-case $O(S^2)$ time complexity, bounded by the maximum number of service-to-service requirements to check on S services. If all services are eventually migrated, our solution is still worst-case exp-time $O(N^S)$, over an infrastructure with N nodes. More likely, as we will epitomise in Sect. 4, a new placement will be determined only for a lower number s of services $s < S$, leading to a $O(N^s) < O(N^S)$ time complexity. For instance, when only one service will be migrated, our approach will be worst-case linear-time $O(N)$ and, when only one service-to-service interaction will trigger migration, it will be worst-case quadratic-time $O(N^2)$.

4 Experimental Results

In this section, we run FogBrain against increasing infrastructure sizes to assess the scalability of our continuous reasoning approach to support runtime application management.

Dataset. In all experiments, the application to be placed is the one of Fig. 1. Target infrastructures replicate the infrastructure of Fig. 3 for a number $R \in \{2, 10, 20, 100, 200\}$ of times, and fully connect nodes with suitable links[3]. Finally, infrastructures feature a single smartphone and a single access point capable of reaching out the VR Viewer needed by the application.

Execution Environment. We run FogBrain in SWI-Prolog[4] 64-bits (v. 8.0.3) on a commodity laptop featuring Windows 10, an Intel i5-6200U CPU (2.30 GHz) and 8 GB of RAM. We rely upon the `time/1` meta-predicate to count the inferences needed by `reasoningStep/5` (line 46) and by `placement/2` (line 49). Counting inferences (instead of only measuring time) permits to assess the performance of continuous reasoning against a machine-independent metric. Finally, we do not consider the inferences needed to load infrastructure data, which we assume can be periodically and incrementally updated by an infrastructure monitoring tool.

Experiments. For all different sizes of the infrastructure we perform the following:

(1) a run of FogBrain to find a first deployment of the application,

(2) a run of FogBrain where infrastructure changes led to no need for migration,

(3a) a run of FogBrain where the Cloud node exploited for deployment features too low hardware resources (viz. 0) versus a run of `placement/2` over the same infrastructure, and

(3b) a run of FogBrain where, instead, the link between the Cloud node and the cabinet server exploited for deployment in the infrastructure features too high latency (viz. 1350 ms) versus a run of `placement/2` over the same infrastructure.

Discussion. Table 1 shows the results of all experiments (1), (2), (3a) and (3b). It is worth noting that determining a first placement (1) requires an exponentially increasing number of inferences as the infrastructure size grows. In our settings, this step requires 1 to 2 seconds to find a valid first placement with 2000 infrastructure nodes, which is a tolerable amount of time when the application is not running yet. However, when it comes to runtime application management, reducing the number of required inferences and, consequently, decision-making times, is crucial to avoid prolonged application performance degradation. As per Table 1, using continuous reasoning to analyse infrastructure changes that do not trigger migrations (2) requires a constant number of inferences, independently of the infrastructure size. In our settings, this corresponds to negligible execution times for all considered infrastructure sizes. This happens because the time complexity of the performed check is at most $O(S^2)$ for an application with S services (as discussed in Sect. 3) and S is constant (viz. 3) throughout the experiments.

Table 1 compares the number of inferences needed to react to (3a) and (3b) with continuous reasoning (i.e. CR Inf.s) to the number of inferences needed to react to them by computing a whole new placement (i.e. No CR Inf.s) using `placement/2`. Table 1 also reports inference speed-ups (viz. (No CR Inf.s / CR Inf.s)) achieved by continuous reasoning for (3a) and (3b). The speed-up for (3a) is over 5500× with an infrastructure of 1000 nodes[5]. Analogously, in the case of a single link failure (3b), which requires migrating two services and all service-to-service allocations, the speed-up is over 95× with an infrastructure of 1000 nodes. It is worth noting that in both (3a) and (3b) FogBrain starts speeding up

[3]The Python code which can be used to generate infrastructures of arbitrary sizes by replicating the base module is available at: `https://github.com/di-unipi-socc/fogbrain/blob/master/infrastructure_builder/builder.py`

[4]SWI-Prolog available at: `https://www.swi-prolog.org/`

[5]The number of CR inferences for (3a) is constant due to the Prolog ordering of the clauses of `node/4` in the example, which always leads to finding a new placement on a node other than the Cloud node in 369 inferences. Anyway, as discussed at the end of Sect. 3, in the worst case, the number of CR inferences increases at most linearly in the number of infrastructure nodes when migrating a single service.

with at least 10 nodes since, for smaller infrastructure sizes, analysing the current deployment conditions requires a number of inferences greater than computing a placement anew.

Table 1: Experimental results.

Nodes	(1) Deploy	(2) No migration	(3a) Cloud node failure			(3b) Cloud-cabinet link failure		
	Inferences	Inferences	CR Inf.s	No CR Inf.s	Speed-up	CR Inf.s	No CR Inf.s	Speed-up
5	457	120	369	300	0.8×	791	517	0.7×
10	883	120	369	695	1.9×	743	926	1.2×
50	7891	120	369	7455	20×	1575	7934	5×
100	25651	120	369	24905	67×	2615	25694	10×
500	527731	120	369	524505	1421×	10935	527774	48×
1000	2055331	120	369	2049005	5553×	21335	2055374	96×
1500	4582931	120	369	4573505	12394×	31735	4582974	144×
2000	8110531	120	369	8098005	21946×	42135	8110574	192×

Overall, the usage of a continuous reasoning approach to support application placement decision-making shows very promising results in our experiments as it substantially reduces the time needed to make decisions at runtime, in the likely situation where only a portion of the deployed application services cannot currently satisfy its requirements.

5 Related Work

In the past, much literature has focussed on the placement of application services to physical servers in Cloud datacentres [19], only a few of which (e.g. [14, 25]) employing a declarative approach. However, managing applications over the Cloud-IoT continuum introduces new peculiar challenges, mainly due to infrastructure scale and heterogeneity, need for QoS-awareness, dynamicity and support to interactions with the IoT, rarely considered in Cloud-only scenarios. Next, we briefly summarise the state of the art in the field of Cloud-IoT multi-service application placement and management, referring the readers to our recent survey [4] for further details.

Among the first proposals investigating the peculiarities of Cloud-IoT application placement, [13] proposed a simple search algorithm to determine an eligible deployment of (multi-service) applications to tree-like Cloud-IoT infrastructures, open-sourced in the iFogSim Java prototype. Building on top of iFogSim, various works tried to optimise different metrics, e.g. service delivery deadlines [16], load-balancing [23], or client-server distances [12]. In our previous work [2, 5], we proved NP-hardness of the placement problem, and we devised a backtracking strategy to determine context-, QoS- and cost-aware placements of multiservice applications to Cloud-IoT infrastructures, also employing genetic algorithms to speed up the search [3]. Based on our work and on the related FogTorchΠ Java prototype, [24] focussed on minimising application response times, while [8] proposed a strategy for Cloud-IoT task offloading. Very recently, we exploited logic programming to assess the security and trust levels of application placements [10], and to determine the placement and network routing of Virtual Network Function chains in Cloud-IoT scenarios [6]. To the best of our knowledge, no other previous work tackling the problem of application placement to Cloud-IoT infrastructures relies on declarative programming solutions. Besides, no previous work proposes continuous reasoning approaches to tame the exp-time worst-case complexity of such a problem, nor to solve it incrementally at application management time.

Finally, proposals exists for simulating application placements and management policies in Cloud-IoT scenarios [17], e.g. YAFS [15], EdgeCloudSim [21], and iFogSim [13] itself. Recently, we also worked on simulating the management of CISCO FogDirector-enabled infrastructures and opensourced

the FogDirSim prototype [11]. Those simulators were mainly used to assess static placements, or dynamic performances of simple management policies (e.g. random, first-fit, best-fit), reasoning on the whole infrastructure and application status when facing infrastructure changes.

Summing up, to the best of our knowledge, this work is the first proposing a declarative continuous reasoning solution to support placement decisions during runtime management of multi-service applications over Cloud-IoT infrastructures.

6 Concluding Remarks

In this article, we presented a novel declarative continuous reasoning methodology, and its prototype FogBrain, to support runtime management decision-making concerning the placement of next-gen multi-service applications to Cloud-IoT infrastructures. FogBrain was assessed over a lifelike example at varying infrastructure sizes, from small-scale to large-scale. Limiting placement decisions only to services affected by the last infrastructure variations via continuous reasoning has shown considerable average speedups (i.e. $\geqslant 2500\times$). Overall, FogBrain represents the core of a continuous reasoner to support Cloud-IoT application management which, being declarative, is concise (125 single lines of code), and easier to understand and extend so to account for new emerging needs, compared to existing procedural solutions ($\geqslant$ 1000s lines of code).

As future work, we plan to extend the continuous reasoning capabilities of FogBrain to also handle changes in the application topology (i.e. addition/removal of services) or requirements (e.g. security policies), to support other runtime management decisions (e.g. application scaling, service adaptation), and to exploit a cost-model and heuristic algorithms to determine optimal eligible placements, possibly along with constraint logic programming and incremental tabling. Besides, we intend to include the possibility to obtain textual explanations on *why* a certain management decision was taken by FogBrain and to exploit probabilistic logic programming to simulate infrastructure variations as per historical monitored data. Last, but not least, we plan to assess FogBrain over other use cases and in testbed settings over actual multi-service applications.

Acknowledgements. Work partly supported by the projects "*DECLWARE*"(PRA_2018_66), funded by the University of Pisa, Italy, and "*GIÒ*", funded by the Department of Computer Science, University of Pisa, Italy.

References

[1] Paolo Bellavista, Javier Berrocal, Antonio Corradi, Sajal K. Das, Luca Foschini & Alessandro Zanni (2019): *A survey on fog computing for the Internet of Things*. *Pervasive Mob. Comp.* 52, pp. 71 – 99, doi:10.1016/j.pmcj.2018.12.007.

[2] Antonio Brogi & Stefano Forti (2017): *QoS-Aware Deployment of IoT Applications Through the Fog*. *IEEE Internet of Things Journal* 4(5), pp. 1185–1192, doi:10.1109/JIOT.2017.2701408.

[3] Antonio Brogi, Stefano Forti, Carlos Guerrero & Isaac Lera (2019): *Meet Genetic Algorithms in Monte Carlo: Optimised Placement of Multi-Service Applications in the Fog*. In: *EDGE 2019*, pp. 13–17, doi:10.1109/EDGE.2019.00016.

[4] Antonio Brogi, Stefano Forti, Carlos Guerrero & Isaac Lera (2020): *How to Place Your Apps in the Fog - State of the Art and Open Challenges*. *Softw. Pract. Exp.* 50(5), pp. 719–740, doi:10.1002/spe.2766.

[5] Antonio Brogi, Stefano Forti & Ahmad Ibrahim (2019): *Predictive Analysis to Support Fog Application Deployment*. In: *Fog and Edge Computing: Principles and Paradigms*, chapter 9, Wiley, pp. 191–222, doi:10.1002/9781119525080.ch9.

[6] Antonio Brogi, Stefano Forti & Federica Paganelli (2019): *Probabilistic QoS-aware Placement of VNF chains at the Edge*. *CoRR* abs/1906.00197. Available at `http://arxiv.org/abs/1906.00197`.

[7] Valeria Cardellini, Vincenzo Grassi, Francesco Lo Presti & Matteo Nardelli (2017): *Optimal Operator Replication and Placement for Distributed Stream Processing Systems*. SIGMETRICS Perf. Eval. Rev. 44(4), pp. 11–22, doi:10.1145/3092819.3092823.

[8] Vincenzo De Maio & Ivona Brandic (2018): *First Hop Mobile Offloading of DAG Computations*. In: *CCGRID 2018*, pp. 83–92, doi:10.1109/CCGRID.2018.00023.

[9] Dino Distefano, Manuel Fähndrich, Francesco Logozzo & Peter W. O'Hearn (2019): *Scaling Static Analyses at Facebook*. Commun. ACM 62(8), p. 6270, doi:10.1145/3338112.

[10] Stefano Forti, Gian-Luigi Ferrari & Antonio Brogi (2020): *Secure Cloud-Edge Deployments, with Trust*. Future Gener. Comput. Syst. 102, pp. 775–788, doi:10.1016/j.future.2019.08.020.

[11] Stefano Forti, Alessandro Pagiaro & Antonio Brogi (2020): *Simulating FogDirector Application Management*. Simul. Model. Pract. Theory 101(102021), pp. 1–18, doi:10.1016/j.simpat.2019.102021.

[12] Carlos Guerrero, Isaac Lera & Carlos Juiz (2019): *A lightweight decentralized service placement policy for performance optimization in fog computing*. J. Ambient Intell. Humaniz. Comput. 10, pp. 2435–2452, doi:10.1007/s12652-018-0914-0.

[13] Harshit Gupta, Amir Vahid Dastjerdi, Soumya K Ghosh & Rajkumar Buyya (2017): *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), pp. 1275–1296, doi:10.1002/spe.2509.

[14] Serdar Kadioglu, Mike Colena & Samir Sebbah (2016): *Heterogeneous resource allocation in Cloud Management*. In: *NCA 2016*, pp. 35–38, doi:10.1109/NCA.2016.7778589.

[15] Isaac Lera, Carlos Guerrero & Carlos Juiz (2019): *YAFS: A Simulator for IoT Scenarios in Fog Computing*. IEEE Access 7, pp. 91745–91758, doi:10.1109/ACCESS.2019.2927895.

[16] Redowan Mahmud, Kotagiri Ramamohanarao & Rajkumar Buyya (2018): *Latency-aware Application Module Management for Fog Computing Environments*. ACM Trans. Internet Techn. 19(1), pp. 9:1–9:21, doi:10.1145/3186592.

[17] Spiridoula V Margariti, Vassilios V Dimakopoulos & Georgios Tsoumanis (2020): *Modeling and Simulation Tools for Fog Computing–A Comprehensive Survey from a Cost Perspective*. Future Internet 12(5), p. 89, doi:10.3390/fi12050089.

[18] Peter W. O'Hearn (2018): *Continuous Reasoning: Scaling the Impact of Formal Methods*. In: *LICS 2018*, pp. 13–25, doi:10.1145/3209108.3209109.

[19] Ilia Pietri & Rizos Sakellariou (2016): *Mapping virtual machines onto physical machines in cloud computing: A survey*. ACM Comput. Surv. 49(3), pp. 1–30, doi:10.1145/2983575.

[20] Yehonathan Sharvit (2019): *A new way of blogging about Prolog*. `http://blog.klipse.tech/prolog/2019/01/01/blog-prolog.html`. Last accessed: July 2020.

[21] Cagatay Sonmez, Atay Ozgovde & Cem Ersoy (2018): *EdgeCloudSim: An environment for performance evaluation of Edge Computing systems*. Trans. Emerg. Telecommun. Technol. 29(e3493), doi:10.1002/ett.3493.

[22] Massimo Villari, Maria Fazio, Schahram Dustdar, Omer Rana, Devki Nandan Jha & Rajiv Ranjan (2019): *Osmosis: The Osmotic Computing Platform for Microelements in the Cloud, Edge, and Internet of Things*. IEEE Computer 52(8), pp. 14–26, doi:10.1109/MC.2018.2888767.

[23] S. Wang, M. Zafer & K. K. Leung (2017): *Online Placement of Multi-Component Applications in Edge Computing Environments*. IEEE Access 5, pp. 2514–2533, doi:10.1109/ACCESS.2017.2665971.

[24] Ye Xia, Xavier Etchevers, Loïc Letondeur, Thierry Coupaye & Frédéric Desprez (2018): *Combining hardware nodes and software components ordering-based heuristics for optimizing the placement of distributed IoT applications in the fog*. In: *ACM SAC*, pp. 751–760, doi:10.1145/3167132.3167215.

[25] Qin Yin, Adrian Schüpbach, Justin Cappos, Andrew Baumann & Timothy Roscoe (2009): *Rhizoma: a runtime for self-deploying, self-managing overlays*. In: *Middleware 2009*, pp. 184–204, doi:10.1007/978-3-642-10445-9˙10.