big data and cognitive computing

Article

LPaaS as Micro-Intelligence: Enhancing IoT with Symbolic Reasoning

Roberta Calegari [1,*], Giovanni Ciatto [2], Stefano Mariani [3], Enrico Denti [1] and Andrea Omicini [2]

1. Dipartimento di Informatica–Scienza e Ingegneria (DISI) ALMA MATER STUDIORUM–Università di Bologna, 40136 Bologna, Italy; enrico.denti@unibo.it
2. Dipartimento di Informatica–Scienza e Ingegneria (DISI) ALMA MATER STUDIORUM–Università di Bologna, 47521 Cesena, Italy; giovanni.ciatto@unibo.it (G.C.); andrea.omicini@unibo.it (A.O.)
3. Dipartimento di Scienze e Metodi dell'Ingegneria (DSMI) Università degli Studi di Modena e Reggio Emilia, 42122 Modena, Italy; stefano.mariani@unimore.it

* Correspondence: roberta.calegari@unibo.it; Tel.: +39-051-209-3274

Received: 15 July 2018; Accepted: 31 July 2018; Published: 3 August 2018

Abstract: In the era of Big Data and IoT, successful systems have to be designed to discover, store, process, learn, analyse, and predict from a massive amount of data—in short, they have to behave intelligently. Despite the success of non-symbolic techniques such as deep learning, symbolic approaches to machine intelligence still have a role to play in order to achieve key properties such as observability, explainability, and accountability. In this paper we focus on logic programming (LP), and advocate its role as a provider of symbolic reasoning capabilities in IoT scenarios, suitably complementing non-symbolic ones. In particular, we show how its re-interpretation in terms of LPaaS (Logic Programming as a Service) can work as an enabling technology for *distributed situated intelligence*. A possible example of *hybrid reasoning*—where symbolic and non-symbolic techniques fruitfully combine to produce intelligent behaviour—is presented, demonstrating how LPaaS could work in a smart energy grid scenario.

Keywords: Logic Programming as a Service; IoT; symbolic reasoning

1. Introduction

According to reference [1], the *Internet of Things* (IoT) can be defined as

"made out of networked sensors and smart objects whose purpose is to measure/control/operate on an environment in such a way to make it intelligent, *usable, programmable, and capable of providing useful services to humans."*

This definition calls for *ubiquitous intelligence* where everyday physical objects should be able to join an IoT network where software components are required to behave intelligently—that is, for instance, understanding their own goals as well as the context where they operate. Devices are thus required to "understand each other, learn and understand situations, and understand us" [2]. This is the path to realise the *Internet of Intelligent Things* (IoIT) vision [3,4].

But what does it mean for devices to be *intelligent*? Historically, two families of techniques have been developed to answer the question, mostly independently since they are traditionally viewed as being antithetic, namely, *symbolic* and *non-symbolic* approaches to artificial intelligence (AI).

Symbolic techniques are desirable for representing domains with highly-structured knowledge. They feature a sound theoretical foundation, many years of practice, and human interpretability. Within the realm of AI, for instance, the explainability of symbolic knowledge is viewed as one of the defining factors that distinguish logic-based representations from statistical or neural ones [5], hence often said to produce "white-box" models. As a paramount example, *Logic Programming* (LP, henceforth) is a declarative programming technique that leverages the idea of logic inference to structure programs as logic theories and computations as proofs of theorems. So far, it has had little impact on the IoT, mostly because of the lack of generally applicable techniques [6]. Also, one of its drawbacks is requiring a precise understanding of the application domain, so as to encode inference rules as logic clauses.

Non-symbolic techniques mostly emerged from the early failures of symbolic ones, as in the case of Brooks' approach to robotics [7,8]. Nonetheless, nowadays the most significant examples of non-symbolic approaches are represented by predominant *machine learning* (ML) techniques, such as *deep learning* [9], and represent the de-facto standard for emulating intelligence in IoT deployments: sensor devices collect data, a central hub (usually, in the cloud) analyses information according to a data processing pipeline, and then commands for actuators are dispatched based on the information inferred from the data. Advancements in computational power, data storage, and parallelisation, in combination with methodological advances in applying ML algorithms, are contributing to the uptake of these approaches witnessed in recent years [9]. However, since they require many examples to actually learn general models of the application domain [10,11], these techniques tend to be computationally expensive–especially for resource-constrained scenarios like the IoT–and suffer from the so-called cold-start problem: at the beginning of a new deployment, there are no data available for training models—let alone "big data". Also, many specific ML algorithms, for instance, those requiring neural networks or deep learning, lack interpretability ("*why* did my model make that prediction?")–accordingly, they are often known as "black-box" models–, which is now becoming a crucial issue, as witnessed by recent "transparency initiatives" of organisations such as the Association for Computing Machinery (ACM) and institutions such as the EU [12,13]. Indeed, the former emphasises the fundamental role that algorithms and data analytics in general play in modern decision-making, while advocating the need for explainability of such a decision-making and accountability of institutions and firms relying on it; the latter is a project started by the European Commission with the explicit aim of gathering scientific evidence about the presence of bias in algorithms exploited in many application areas, for instance, in filtering access to information online.

Recently, big industry players (such as IBM, Amazon, and Microsoft) started to tie up ML (and non-symbolic techniques in general) with IoT, opening the way to new intriguing scenarios [11,14]. We believe that the complexity of scenarios nowadays, especially those featuring different problems at different *scales*, would greatly benefit from a hybrid approach where symbolic and non-symbolic approaches co-exist and *complement* each other. Combining symbolic reasoning with non-symbolic approaches may help to address the fundamental challenges of reasoning, hierarchical representations, transfer learning, and robustness in the face of adversarial examples and interpretability [15–17]. These needs are highly interdependent and stem from the push towards trustworthy computing applied in the context of the IoT paradigm [17]. It is worth remarking that our vision does not propose to cast out non-symbolic techniques as ML ones; rather, it proposes to synergically exploit both symbolic and non-symbolic approaches, since they better deal with different requirements. For instance, reasoning over symbolic knowledge bases allows, among others, consistency checking (i.e., detecting contradictions between facts or statements), classification (i.e., generating taxonomies), and other forms of deductive inference (i.e., revealing new, implicit knowledge given a set of facts).

As discussed in Section 5, we envision IoT systems benefit by mitigating some of the issues experienced by non-symbolic approaches and by enabling novel forms of distributed and local reasoning, adding all the symbolic techniques advantages to the system. In this way models become "grey-boxes" in the spirit of reference [2]—that is, they are partially understandable.

The scenario depicted above implies serious challenges from a *software engineering* perspective, starting from a careful design of both the ML pipeline and the logic inference engine, separately, and proceeding with their complementary exploitation. In this paper we focus on the latter—that is, on how traditional LP can be geared toward this future of hybrid reasoning. This issue does not merely concern the features that the LP software component is meant to provide, but also the way in which it should be developed in order to achieve it, i.e., the interoperability of deployment environments, the scalability of software delivery, and the efficiency of software development [18].

For the reason above, we discuss *Logic Programming as a Service* (LPaaS) [19] both as an enabling technology and in terms of the way it is developed, and advocate its role as a provider of symbolic reasoning techniques in IoT scenarios, complementing non-symbolic ones. To this end, we move from the LPaaS re-interpretation of *distributed logic programming* [20] in the IoT era, which re-casts logic programming as a *microservice* that is easily deployable in compliance with the current agile software development and continuous delivery best practices. The idea of exploiting LP for the IoIT moves from the fact that LP, by definition, should be the paradigm of choice for endowing software with some form of intelligence, such as inference of novel data, deliberation of actions, etc.

In particular, we extend our previous work [21] by delving deeper into the benefits of combining LPaaS with non-symbolic approaches in IoT scenarios by discussing an envisioned case study regarding energy grid operations. Accordingly, we recall the main software engineering challenges in the IoT context highlighting the benefits of such a mixed approach (Section 2), and then introduce the LPaaS vision and architecture (Section 3), and the RESTful design alongside the implementation process (Section 4), emphasising the development process and the microservice nature of LPaaS. Then, we present the energy grid use case to illustrate the need for and the expected benefits of LPaaS (Section 5), and compare LPaaS with other approaches to distribute intelligence in IoT deployments (Section 6). Finally, conclusions are drawn in Section 8.

2. Intelligence in IoT: Techniques, Challenges, and Opportunities

Making sense of IoT data —that is, gaining value from them —has been a hot topic and challenge of IoT research since its early days. The quest for collecting and sharing context information from connected things has been addressed in earlier research [22,23], often by architecting ad-hoc solutions to address a specific aspect, such as device management, context information collection and sharing, context-awareness, interoperability, etc. [24]. However, there is currently no single and standardised solution that covers all such challenges [24]: new models and technologies are required in order to face the distributed intelligence issues.

A critical issue, in particular, concerns knowledge consistency—including updating business logic rules or policies automatically and possibly learning from past experiences. This is particularly true when the IoT platform is cloud-centric, and each device might control thousands of things to be updated synchronously and consistently.

The main Artificial Intelligence (AI) techniques exploited so far to deal with these issues belong to two main categories—namely, *symbolic* and *non-symbolic* techniques. From a software engineering (SE) perspective, engineering models and technologies to distribute intelligence in such environments means properly facing IoT challenges in terms of *flexibility*, *interoperability*, and *scalability*. Accordingly, in this section, we first discuss the two main AI techniques for intelligence (Section 2.1), highlight the SE challenges of IoT (Section 2.2), and finally, propose the LPaaS vision as a possible answer to the above challenges and opportunities (Section 2.3).

2.1. Symbolic vs. Non-Symbolic Techniques: Features & Synergies

Both researchers and industry players have recently started to tie up ML (and non-symbolic techniques in general) for spreading intelligence and learning capabilities in the IoT [9,11,14]. Non-symbolic techniques are based on statistical methods, in which intelligence is taken as an emergent property of the system. The intelligent behaviour is commonly formulated as an optimisation problem, whose solutions lead to behaviour that resembles intelligence. Among their many features, it is worth emphasising [25,26]:

- The capability to learn patterns or regularities from data, without (or, with very little) a-priori knowledge;

- The ability to generalise beyond the training domain of knowledge; and
- The possibility of run-time tuning of models to make more timely and accurate predictions, as well as to ease deployment.

However, non-symbolic approaches are often computationally expensive and lack some typical features of symbolic approaches. First, their implicit black-box nature can make them unsuitable to domains where verifiability is important [27]; moreover, their inability to reason at an abstract level makes it difficult to implement high-level cognitive functions, such as transfer learning, analogical reasoning, and hypothesis-based reasoning [15]. This is why several works in the last decade have focussed on extracting symbolic rules [28] from the knowledge implicit in a black-box model.

Symbolic approaches, on the other hand, represent things within a domain of knowledge through symbols, combine symbols into symbol expressions, and manipulate the latter, possibly through inferential processes. Symbolic representations feature two important properties that we intend to focus on, among the many [28]:

- Observability, explainability and accountability, in the sense of human readability, transparency of inference rules, and opportunities to provide explanations about the outcomes of the inference process;
- Malleability, in the sense of the capability to easily update rules and policies at run-time in an automatic way; and
- Composability, in the sense that more complex representations can be obtained by composing basic symbols with appropriate rules, while maintaining basic symbols that are recognisable in complex representations—which is useful for producing new symbols through inference rules.

Moreover, reasoning on such knowledge bases makes it possible to deal with issues like consistency checking (i.e., detecting contradictions between facts or statements), classification (i.e., generating taxonomies intended as representation of knowledge with the subdivision of phenomena into classes [29]), and other forms of deductive inference (i.e., revealing new, implicit knowledge given a set of facts). However, several issues arise with symbolic approaches, too. Among these, it is worth mentioning performance, which is often good only in narrow domains facing specific problems, and the "no types" approach, that makes them distant to mainstream programming paradigms.

Although symbolic approaches and LP languages represent, in principle, natural candidates for injecting intelligence within computational systems [30], the adoption of such approaches in pervasive contexts has been historically hindered by technological obstacles—efficiency and integration issues—as well as by some cultural resistance towards LP-based approaches outside the academy. For a survey on practical applications developed over the years and the related benefits, we forward the reader to references [31–34].

2.2. *Software Engineering Challenges*

From a SE perspective, no consolidated set of best practices has emerged so far in the IoT world [1]; indeed, properly engineering such a new generation of scalable, highly-reactive, (often) resource-constrained software systems is still a challenge, for both symbolic and non-symbolic approaches. There is a need for rapid, reliable, scalable, and evolvable processes (and guidelines), specifically tailored to the IoT peculiar features; for instance, interoperability is a major issue in the realm of so many diverse technologies, especially considering the increasing pace of change in technology. As the number of devices involved in IoT deployment grows, delivering software updates and handling scalability in a timely manner becomes crucial, too. Flexibility of deployments is also a major issue, in order to avoid developers having the overhead of heavily re-configuring an existing IoT solution for a slightly different application.

With agile methods bringing significant contributions with respect to these requirements [35], service-orientation and agile deployment paradigms have recently culminated in a new architectural style referred to as *microservices* [36]. Moving from *Service-Oriented Architecture* (SOA) [37], LPaaS is specifically designed and developed according to these best practices. A microservice can be seen as a specialisation of a service-based system, which consists of very fine-grained independent services and is rooted in the decentralisation of control and management, lightweight communication mechanisms, and technological heterogeneity (and interoperability). The software development process has moved on from well-defined requirements, emphasising the need to support fast-changing requirements, fast-paced release cycles, as well as the scalability of deployments.

A microservice should be a small and coherent unit of functionality (with a bounded context) so as to gain benefits such as increased application resiliency, efficient scalability, as well as fast and independent deployment. More generally, with the very notion of service, functionalities should be offered via a uniform and platform-agnostic interface so as to achieve interoperability among devices and communication protocols. Each service is thus supposed to be self-contained, composable, technology-neutral, and loosely coupled as well as to allow for location transparency.

2.3. LPaaS Approach to Micro-Intelligence

The aim of LPaaS is to provide a complementary solution to non-symbolic techniques in order to achieve distributed intelligence while adhering to the major SE requirements posed by the agile paradigm and microservices in the IoT context. According to the *micro-intelligence* vision, LPaaS means to provide intelligence at the *things* level, as suggested in [24,38,39], highlighting the need for different scales of intelligence. The *intelligence of things* is then provided by gathering information and inference processes closer to the devices, towards the edges of computing networks, by enabling local symbolic reasoning (at the scale of "small data") to co-exist with global non-symbolic reasoning (alongside the scale of Big Data, usually in the cloud). The micro-level feature aims to highlighting and reminding such a peculiarity of intelligence; very small chunks of intelligence, spread all over the system, are capable of enabling the individual intelligence of devices of any sort, promoting interoperation and smart coordination among diverse entities.

As depicted in Figure 1, LPaaS makes symbolic approaches suitable for the IoT domain by tackling the SE challenges which traditionally hindered its exploitation. In particular, the re-interpretation of the LP paradigm under the microservices architecture paves the way towards designing intelligence according to interoperable, reliable, composable, and scalable functionalities, while enabling observability and understandability. In the following section, we present the LPaaS model and architecture as well as the software engineering process adopted to deliver it.

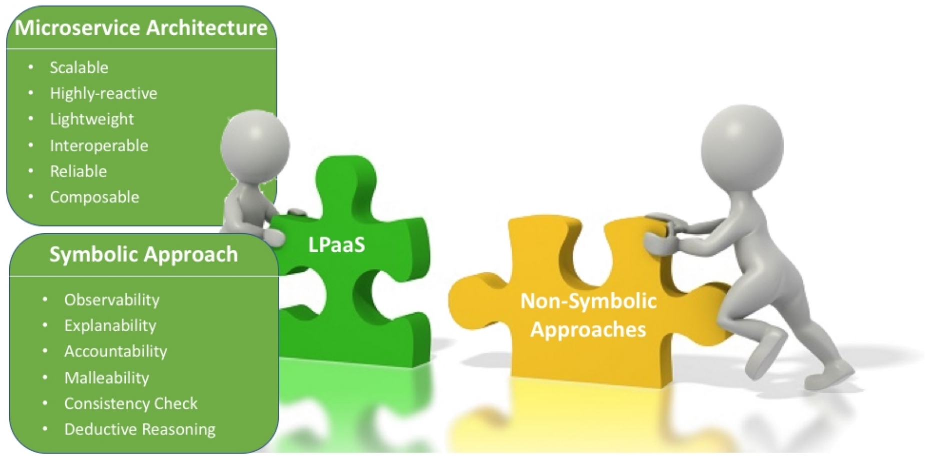


Figure 1. Logic Programming as a Service (LPaaS) contribution with respect to *(i)* Internet of Things (IoT) software engineering (SE) challenges and *(ii)* complementing non-symbolic approaches.

3. LPaaS: Model and Architecture

LPaaS is based on the idea of providing an inference engine in the form of a service—library service, middleware service, network service, etc.—leveraging the power of LP resolution [20]. For this purpose, we re-interpret LP from the service perspective as a means of *spreading intelligence* in a pervasive IoT system. LPaaS [19] was, in fact, specifically designed to enable and take advantage of *situatedness*—the property of making computations sensitive to the local context—for the target application context, so that it is possible to reason efficiently for data that are local to situated components. Not by chance, situatedness is a crucial feature for most IoT applications—although it is not always explicitly acknowledged, but comes as a consequence of context-awareness. Being based on the LP model, diverse computational models can be tailored to the local needs, exploiting LP extensions explicitly aimed at pervasive systems such as *labelled variables systems* [40,41]. Labelled variables systems are meant to deal with domain-specific knowledge so that diverse computational models, tailored to the specific needs of situated components, can fruitfully coexist side by side. More precisely, by labelling variables and suitably defining a computational model for label elaboration and propagation, a domain-specific computational model can be added to traditional LP, thus enabling diverse computational models to interact within a logic-based framework.

The design of LPaaS in terms of *situated service* moves along three main directions: *(i)* devising out a modular architecture—to guarantee a clear separation of concerns and to reduce the overall system complexity; *(ii)* defining the standard APIs—to guarantee a standardised network-addressable entry point and to provide data contracts; and *(iii)* adopting a *continuous delivery* pipeline—to represent the automation needed to provide a continuous release of value to the end user. All of the above is in the spirit of agile software development and aims to follow microservice best practices and architectural principles.

3.1. The Situated Nature of LPaaS

The resolution process [42] remains a staple in LPaaS, yet it is re-contextualised according to the situated nature of the specific LP service. Given the precise spatial, temporal, and general context within which the service is operating when the resolution process starts, the process follows the usual rules of resolution. At the same time, this context is exactly what can make resolution come up with different solutions to the same queries.

Making LP aware of space–time essentially means making the resolution process sensitive *(i)* to the passing of time, and *(ii)* to the topology of space—that is, being aware of the existence of different points in time/space and of the possibility of moving between them. It is worth remarking that spatio-temporal awareness is nowadays a strict requirement for the vast majority of IoT applications, where periodic or scheduled tasks and location-based services are a norm rather than an exception [22,43]. As an illustrative example, consider the case of a number of LPaaS inference engines distributed in a smart building. Depending on the sensors available in the surroundings and on the measures they stream to the closest LPaaS node, answers to queries and proofs to goals about complex situations may be different, depending on the specific LPaaS service instance involved. Moreover, the situation perceived as well the proofs to goals may change as time passes, i.e., because measurements from sensors change due to inner dynamics of the environment they live within.

In classical LP, a goal is basically proven against a logic theory that is considered true, independently of space and time—that is, regardless of when and where the resolution process takes place. LPaaS promotes a broader vision—the resolution process becomes sensitive to the time and space dimension. Hence, a goal is demonstrated against a logic theory that is considered true within a (possibly open) *interval of time* and within a *region of space*. Notice that this helps in dealing with the issue of guaranteeing *global* consistency in large-scale, ever-changing scenarios, such as the IoT; with LPaaS, *local* consistency within individual knowledge bases distributed over the network suffices to support local reasoning. Our previous smart building example may help to clarify the above. Given the existence of many different instances of the LPaaS service, distributed in the area covered by the building and possibly available at different times in its lifespan, it makes no sense to talk about, for example, the global consistency of the logic theory stored in LPaaS—basically because there is no actual use for it. What is meaningful, on the contrary, is the local consistency and reasoning that is available and "correct" in a precise region of space and time, i.e., in our exemplary scenario, this refers to a given room in a given day.

In principle, the resolution process could also *span* and *move* over time and space. Intuitively, moving back in time means to ask the LPaaS service the solution of a given goal at a certain moment in the past, whereas moving forward in time either means to ask to "predict" the solutions that could become available in the future, or to wait until the goal become provable. Analogously, moving in space in either direction means to consider a different region of validity associated to the logic theory, providing solutions that are situated (hold true) in a given region of space—and only provable therein. We refer the interested reader to [44] for a deeper discussion of the subject.

3.2. LPaaS Interface

Each LP server node exposes its services concurrently to multiple clients via suitable interfaces. The inference engine is expected to implement SLD (Selective Linear Definite) resolution [42]. Like in classical LP, it is configured with a *theory* of axioms—its knowledge base (KB). Unlike classical LP, however, it is also configured with a set of *goals*—the only admissible queries for which a proof can be asked. The reason for this design choice is that micro-intelligence is supposed to address specific local needs that correspond to specific possible queries (situated in space and time), not to provide a general LP inferential engine.

The service is initialised on the server at deployment-time. Once started, it can be accessed by *clients* (querying the KB for logic facts or asking for proofs of admissible goals), *sources* (meant to represent sensors and actuators willing to update the KB with latest measurements and action feedbacks), and *configurators* (privileged agents with the right to dynamically re-configure the service at run-time, subject to proper access credentials) via the corresponding interfaces. Since sources can modify the theory by adding new facts, they are designed as privileged clients (access credentials have to be provided). Specifically, the *client interface* exposes methods for observation and usage, the *source interface* provides methods for adding data (logic facts) to the KB, and the *configurator interface* accounts for service configuration, as depicted in Figure 2a. Figure 2b shows an example of use of the source interface. There, the LPaaS service has been deployed on a smart home conditioner to control the temperature of a room. The device autonomously decides when to turn itself on/off based on measurements of two sensors—namely, temperature and presence. Once the service is properly configured through the *configurator interface* (not depicted), the two sensors exploit the *source interface* to add up-to-date facts to the KB, while the user or other devices can access the service exploiting the *client interface*. Details on the interfaces' APIs are discussed in Section 3.3 below.

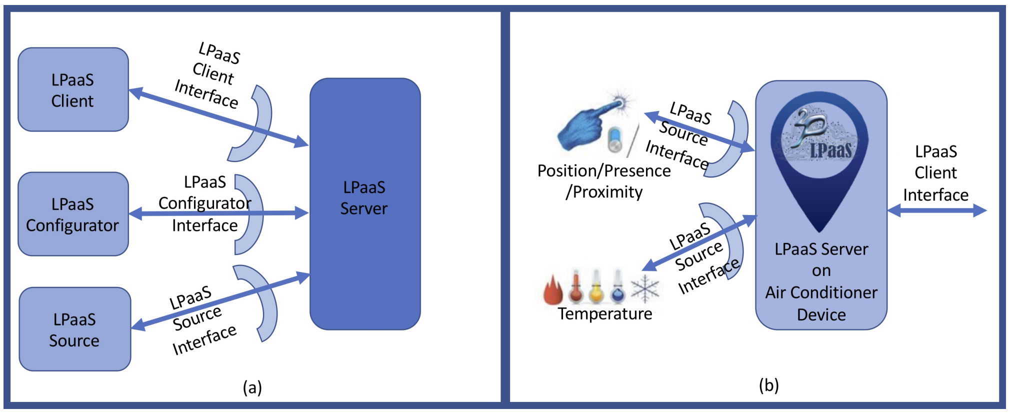


Figure 2. The LPaaS actors and their interfaces (**a**). An example of the use of the source interface (**b**).

As far as the network architecture is concerned, the LPaaS model does not put any constraint on deployment solutions. An LPaaS node (server) can be placed in the cloud, on a IoT device, or in a dedicated gateway component (as in edge/fog computing). Presently, the available implementation provides LPaaS as a RESTful web service developed on a stack of Java-based technologies, which guarantees extreme portability and availability. As a final note, although different LPaaS nodes do not share information—as by design they represent different knowledge that is contextual to where and when they live—they can become aware of each other's existence, so as to, for instance, collectively try to respond to queries involving regions of space—see Section 3.3.

3.3. LPaaS API

The LPaaS API methods are detailed in Table 1 (configurator interface and source interface), and Table 2 (client interface)—adopting the standard Prolog notation for input/output where "+" means that the argument must be instantiated (input argument), "−" that it is an output argument (usually non-instantiated, unless one wants to check for a specific return value), and "?" that it can be either instantiated or not [45].

As far as the logic theory is concerned, a *static* KB is immutable, while a *dynamic* KB can evolve during the server's lifetime, thus implying that clauses have lifetimes too and can be asserted and retracted as needed—for instance, a clause representing the current temperature in a room. The static knowledge base reflects the view of LP based on a static representation of knowledge with the standard LP semantics under the closed-world assumption (CWA) [46]. In IoT scenarios, this possibility is of great interest when dealing with invariants, such as properties or statements, that must be always guaranteed to be true, no matter what. In our model, the dynamic knowledge base which takes the time dimension into account constitutes a more general case than the static one. Accordingly, the static KB predicates can be seen as a simpler subset of the dynamic KB ones, as discussed below.

Client interaction can be either *stateless* or *stateful*. In the former case, each request is self-contained—and thus, always repeats the goal to be proven—while in the latter case, requests are tracked on the server side so that once the goal is set, there is no need to repeat its specification in subsequent requests. It is worth noting that while the service can act simultaneously statefully and statelessly—as it can manage different kinds of request concurrently—the knowledge base is, by its nature, either dynamic or static.

Table 1. LPaaS configurator interface (left) and LPaaS source interface (right).

Configurator Interface	Source Interface
`setConfiguration(+ConfigurationList)`	`addFacts(+Facts)`
`getConfiguration(-ConfigurationList)`	`getFacts(-Facts, template(?Template), ?Timestamp)`
`resetConfiguration()`	
`setTheory(+Theory)`	
`getTheory(-Theory)`	
`setGoals(+GoalList)`	
`getGoals(-GoalList)`	

The LP service offers a set of methods to provide configuration and contextual information about the service and its KB, and a set of methods to query the service for triggering computations and reasoning, and to ask for solutions. In particular, the *configurator interface* makes it possible to set the service configuration, its KB, and the list of admissible goals. The *source interface* allows new facts to be added to the KB, meant to represent sensors and actuators with the latest measurements or feedback actions. The *client interface* makes it possible to query the service about its configuration parameters (stateful/stateless and static/dynamic), the state of the KB, and the admissible goals and facts. Moreover, the interface allows the service to be asked for one solution, N solutions, or all solutions available. These operations are slightly different in case of stateless or stateful requests. In the first case, the `solve` operation is conceptually atomic and self-contained and always has `Goal` as its parameter, whereas in the latter case such a self-containment is not necessary, since the server keeps track of the client state, so the goal can be set only once before the first `solve` request is issued.

Unlike classical LP, LPaaS also provides an extra `solve` operation for dealing with *streams of data* (i.e., stream of solutions), that are often found in IoT contexts. This extra operation is particularly useful when accounting for source clients. Since they are meant to add data to the LPaaS KB dynamically—and *monotonically*—it makes sense to allow clients to get a novel solution to the goal set at each *t* time step (parameter `every(@Time)`), to promptly react to changes in contextual data—such as the temperature of a room. In addition, a family of `solve` predicates with a specific `within(+`*`Time`*`)` argument (intended as server-side local time) prevents the server from being indefinitely busy. If the resolution process does not complete within the given time, the request is cancelled, and a failure response is returned to the client.

Besides these novelties, mostly regarding efficient interaction between the service and its clients, time can affect computations, leading to *time-dependent computations*, and the *temporal validity* of logic theories as well, and thus, of all the individual facts and clauses therein whose validity can be collectively bound to a given time horizon—i.e., being true up to a certain instant in time and no longer after that. Computation requests may then arbitrarily restrict the *temporal scope* of the expected solutions, specifying the temporal bounds of the logical facts and clauses to be considered valid while proving a goal. This is the way in which LPaaS supports temporal situatedness, that is, the ability to be aware of time passing by and the fact that, in dynamic scenarios like the IoT, the knowledge base represents *truth at a given point in time*.

With respect to space situatedness (Section 3.1), the LPaaS server inherently has a notion of its own location, since the service container is physically located somewhere. By specifying the "width" of a region, defined according to some (custom) metric, we can define the service *surroundings*. The key point is that LPaaS can explore its surroundings to discover other LPaaS instances, representing different local knowledge, and forward to them the query looking to expand the solutions that it found by itself. This concept is captured by the `solveNeighborhood/2` primitive that associates the inference process to a region centred in *`Pos`*—if omitted, the server position is considered—and distance is specified by *`Distance`*. Similarly to the time case, the client can also open a *stream of solutions across space*. This could be interpreted, for instance, as widening/narrowing the region to be considered at each cycle of the resolution process. The extra *`Span`* argument in the same primitive satisfies this requirement, specifying how much to widen/narrow the range at each cycle.

Finally, the `reset` primitive clears the resolution process, effectively restarting resolution with no need to reconfigure the service (i.e., select the goal); in turn, the `close` primitive actually ends communication with the server—so that the goal must be re-set in order to restart querying the server.

Table 2. LPaaS client interface.

DYNAMIC KNOWLEDGE BASE	
Stateless	**Stateful**
`getServiceConfiguration(-ConfigList)`	
`getTheory(-Theory, ?Timestamp)`	
`getGoals(-GoalList)`	
`isGoal(+Goal)`	
	`setGoal(template(+Template))`
	`setGoal(index(+Index))`
`solve(+Goal, -Solution, ?Timestamp)`	`solve(-Solution, ?Timestamp)`
`solveN(+Goal, +NSol, -SList, ?TimeStamp)`	`solveN(+NSol, -SolutionList, ?TimeStamp)`
`solveAll(+Goal, -SList, ?TimeStamp)`	`solveAll(-SolutionList, ?TimeStamp)`
`solve(+Goal, -Solution, within(+Time), ?TimeStamp)`	`solve(-Solution, within(+Time), ?TimeStamp)`
`solveN(+Goal, +NSol, -SList, within(+Time), ?TimeStamp)`	`solveN(+NSol, -SList, within(+Time), ?TimeStamp)`
`solveAll(+Goal, -SList, within(+Time), ?TimeStamp)`	`solveAll(-SList, within(+Time), ?TimeStamp)`
`solveAfter(+Goal, +AfterN, -Solution, ?TimeStamp)`	
`solveNAfter(+Goal, +AfterN, +NSol, -SList, ?TimeStamp)`	
`solveAllAfter(+Goal, +AfterN, -SList, ?TimeStamp)`	
	`solve(-Solution, every(+Time), ?TimeStamp)`
	`solveN(+N, -SList, every(+Time), ?TimeStamp)`
	`solveAll(-SList, every(+Time), ?TimeStamp)`
	`pause()`
	`resume()`
`solveNeighborhood(+Goal, -Solution, region(?P, +Space), ?TimeStamp)`	`solveNeighborhood(-Solution, region(?P, +Space), ?TimeStamp)`
`solveNNeighborhood(+Goal, +NSol, -SList, region(?P, +Space), ?TimeStamp)`	`solveNNeighborhood(+NSol, -SList, region(?P, +Space), ?TimeStamp)`
`solveAllNeighborhood(+Goal, -SList, region(?P, +Space), ?TimeStamp)`	`solveAllNeighborhood(-SList, region(?P, +Space), ?TimeStamp)`
	`solveNeighborhood(-SList, region(?P, +Space, +Span), ?TimeStamp)`
	`solveNNeighborhood(-SList, region(?P, +Space, +Span), ?TimeStamp)`
	`solveAllNeighborhood(-SList, region(?P, +Space, +Span), ?TimeStamp)`
`solveNeigh.After(+Goal, +AfterN, -SList, region(?P,+Space,+Span), ?TimeS)`	
`solveNNeigh.After(+Goal, +AfterN, +N, -SList, region(?P,+Space,+Span), ?TimeS)`	
`solveAllNeigh.After(+Goal, +AfterN, -SList, region(?P, +Space, +Span), ?TimeS)`	
`reset()`	
`close()`	

The table reports only methods operating on a dynamic KB that take an additional `Timestamp` argument, expressing the required time validity; static KB methods are analogous without the `Timestamp` parameter: for a full description of the API, we refer the reader to [47].

3.4. LPaaS Architecture

As discussed in reference [47], the LPaaS microservice architecture is composed of three logical units (depicted in Figure 3): the *interface* layer, the *business logic* layer, and the *data store* layer. The interface layer encapsulates the configurator, client, and source interfaces. The business logic wraps the Prolog engine with the purpose of managing incoming requests. The data layer is responsible for managing the knowledge base (KB).

Mapping the logical architecture onto a concrete architecture offers some degrees of freedom. The choice here is to design the whole LPaaS microservice as a composition of *three microservices*—one for each logical unit. In this way, scalability and portability are extended beyond LPaaS inference service, offering a more fine-grained control on deployment—for instance, enabling the effortless relocation or replication of the whole LPaaS, of the sole inference engine (service), or of the KB alone. The choice fits the IoT scenarios well where applications usually have a datastore—for instance, in the context of Big Data (IoT-Big Data), where data are continuously generated, mainly from sensor systems. In such scenarios, the above choice makes it easy to scale the KB microservice in/out individually. Such a modular decomposition also allows for easy and efficient deployment on a network of distributed devices hosting LPaaS.

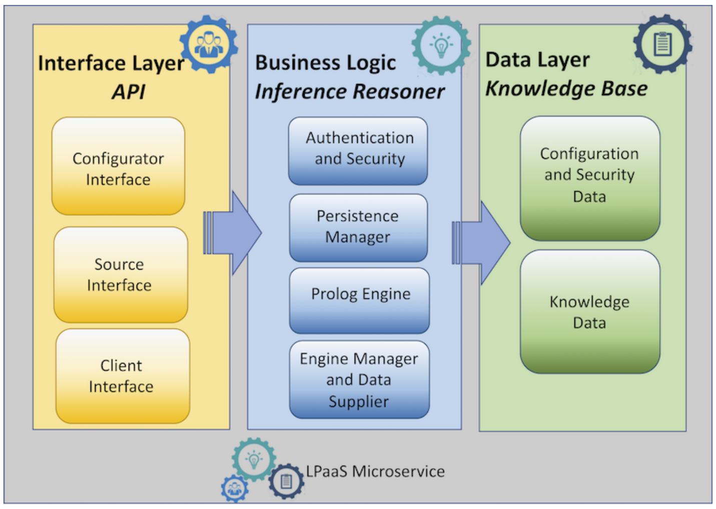


Figure 3. The LPaaS microservice.

Interoperability in LPaaS relies on standard representation formats, namely, JSON (JavaScript Object Notation), and common interaction protocols and best practices, such as the adoption of RESTful APIs, which are nowadays supported by the vast majority of IoT-enabled devices.

4. RESTful LPaaS: Development Process from Design to Deployment

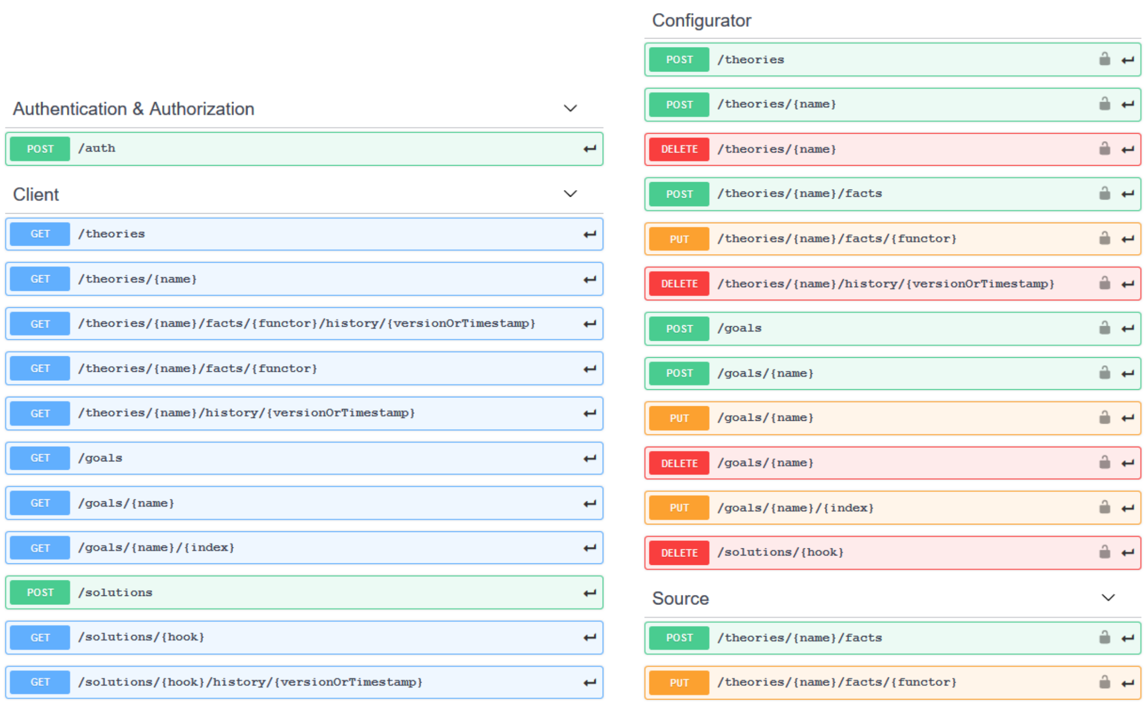
As a first prototype, we chose to realise LPaaS as a web service in compliance with the *Respresentational State Transfer* (REST) architectural style [48], thus re-interpreting the main features of LPaaS and its API (Tables 1 and 2) in terms of *web resources* being accessible through a stateless HTTP-based interface. This allows both clients and LPaaS itself to be implemented with different technologies and programming languages while remaining mutually interoperable.

Of course, in a RESTful implementation, the statefulness disappears to better comply with REST principles. However, it is worth emphasising here that this is just one among the possible implementations of LPaaS; thus, since other design choices could benefit from a stateful approach, the corresponding generic LP API should not be disregarded. For instance, by adopting the SOA architectural style, we could genuinely take statefulness into account while retaining the advantages of leveraging a web-service implementation.

The following sub-sections discuss the RESTful API for LPaaS, the corresponding implementation, and its deployment process.

4.1. RESTful API Design

The three interfaces from Section 3.2 are collapsed within a unified RESTful API exposing the same *resources* to clients possibly playing different *roles*—namely *client*, *source*, or *configurator*. The two latter roles, since they are capable of applying *side-effects* to the service KB, require authentication. There are four main REST resource types provided: `theories`, `goals`, `solutions`, and `authorisation` tokens—as shown in Figure 4, along with their sub-resources and supported operations.



Authentication & Authorization

POST /auth

Client

GET /theories

GET /theories/{name}

GET /theories/{name}/facts/{functor}/history/{versionOrTimestamp}

GET /theories/{name}/facts/{functor}

GET /theories/{name}/history/{versionOrTimestamp}

GET /goals

GET /goals/{name}

GET /goals/{name}/{index}

POST /solutions

GET /solutions/{hook}

GET /solutions/{hook}/history/{versionOrTimestamp}

Configurator

POST /theories

POST /theories/{name}

DELETE /theories/{name}

POST /theories/{name}/facts

PUT /theories/{name}/facts/{functor}

DELETE /theories/{name}/history/{versionOrTimestamp}

POST /goals

POST /goals/{name}

PUT /goals/{name}

DELETE /goals/{name}

PUT /goals/{name}/{index}

DELETE /solutions/{hook}

Source

POST /theories/{name}/facts

PUT /theories/{name}/facts/{functor}

Figure 4. The LPaaS RESTful API. For further details please refer to the LPaaS Home Page [49].

All resources are accessible by specifying the so-called *entry point* after the service base URL, for instance, `www.lpaas-example.it/` *theories*, where clients may exploit the logic service functionalities by issuing HTTP requests on the corresponding entry point. The response contains the requested information (if possible), an acknowledgement, or some link for late access to the result. For instance, users wanting to play some privileged role `POST` their credential to the`/auth` entry point. A comprehensive description of the RESTful entry points and their supported HTTP methods is provided in Table 3, in terms of mapping the LPaaS interface API given in Tables 1 and 2.

Configurators can create theories by `POST`ing a well-formed Prolog theory on the `/theories` entry point. There are multiple *named* theories and, for each theory, many versions can be maintained. According to time situatedness, each version of a theory is valid from the instant it has been `POST`ed (service local time) until either a newer version is `POST`ed, or some modification is made through some other entry point. The specific version, say the one valid at time `t`, of a particular theory `n` can be read by clients via a `GET` operation at the `/theories/n/history/t` entry point. In regard to spatial situatedness, whenever information—i.e., a theory or fact version—is retrieved via a `GET` operation, the response includes a spatial tag defining the position in space where the returned information can be considered valid.

Theory versions make it possible to preserve LP consistency despite the side effects possibly occuring on dynamic KBs; in this way, clients are guaranteed to re-obtain the same solution if the same goal(s) is re-asked on the same theory version.

Table 3. The Respresentational State Transfer (REST)ful reification of LPaaS API.

Legend	
HTTP_METHOD: /resource?queryParam=value Request content Response Content	LPaaS Interface API
***configurator* role**	
POST /theories/*default* Req: *T* Res: -	setTheory(*T*)
PUT /goals/*default* Req: *G* Res: -	setGoals(*G*)
***source* role**	
GET /theories/*default*/facts/*f* Req: - Res: *F*	getFacts(*F*, template(*f*(_)), _)
GET /theories/*default*/facts/f/history/*V* Req: - Res: *F*	getFacts(*F*, template(f(_)), *V*)
POST /theories/*default*/facts Req: *F* Res: -	addFacts(*F*)
***client* role**	

GET /theories/*default*
Req: - Res: *T*
GET /theories/*default*/history/*V*
Req: - Res: *T*
	getTheory(*T*, _)
	getTheory(*T*, *V*)

GET /goals/*default* Req: - Res: *G*	getGoals(*G*)
POST /solutions?limit=1 Req: *G* Res: *S*	solve(*G*, *S*, _)
POST /solutions?limit=1 Req: *G*, *V* Res: *S*	solve(*G*, *S*, *V*)
POST /solutions?limit=*N* Req: *G*, *V* Res: *S*	solveN(*G*, *N*, *S*, *V*)
POST /solutions Req: *G*, *V* Res: *S*	solveAll(*G*, *S*, *V*)
POST /solutions?limit=1&within=*T* Req: *G*, *V* Res: *S*	solve(*G*, *S*, within(*T*), *V*)
POST /solutions?limit=*N*&within=*T* Req: *G*, *V* Res: *S*	solveN(*G*, *N*, *S*, within(*T*), *V*)
POST /solutions?within=*T* Req: *G*, *V* Res: *S*	solveAll(*G*, *S*, within(*T*), *V*)
POST /solutions?skip=*N*&limit=1 Req: *G*, *V* Res: *S*	solveAfter(*G*, *N*, *S*, *V*)

POST /solutions?skip=*N1*&limit=*N2*
Req: *G*, *V* Res: *S*
POST /solutions?skip=*N*
Req: *G*, *V* Res: *S*
	solveNAfter(*G*, *N1*, *N2*, *V*)
	solveAllAfter(*G*, *N*, *S*, *V*)

POST /solutions?every=*DT*&hook=*H* Req: *G*, *V* Res: *H* GET /solutions/*H* Req: - Res: *S*	setGoal(template(*G*)), solve(*S*, every(*DT*), *V*)
POST /solutions?where=*W*&regionSize=*R* Req: *G*, *V* Res: *S*	solveNeighborhood(*G*, *S*, region(*W*, *R*), *V*)
POST /solutions?where=*W*&regionSize=*R* &regionSpan=*DS*&hook=*H* Req: *G*, *V* Res: *H* GET /solutions/*H* Req: - Res: *S*	setGoal(template(*G*)), solveNeighborhood(*G*, *S*, region(*W*, *R*, *DS*), *V*)

4.1.1. Configurator

Other than having the right to create or edit theories, clients playing the *configurator* role can create individual goals or lists of goals (*composed-goals*) by `POST`ing a well-formed conjunction of Prolog terms on the `/goals` entry point, along with a mnemonic name (say *g*). Goals can be lately appended to the list by `POST`ing them on the `/goals/`*g* entry point, while the *i*-th sub-goal of *g* can be replaced by `PUT`ing *g'* on the `/goals/`*g*`/`*i* entry point. Of course, configurators are also authorised to perform any other operation, including the ones available to source or client roles.

4.1.2. Source

Sensors, actuators, or other clients playing the *source* role may also inject (or update) novel information—in the form of logic *facts* wrapped by a proper functor (say *f*)—in a theory *n* by `POST`ing (resp. `PUT`ting) the new information to the `/theories/`*n*`/facts/`*f* entry point. The latest version of a specific fact (*f*) in theory *n* can be read by any client `GET`ting the aforementioned entry point, while the fact version valid at *t* can be obtained querying the entry point, `/theories/`*n*`/facts/`*f*`/history/`*t*.

4.1.3. Client

Any client can get the list of admissible goals by `GET`ting the `/goals/`*g* entry point, and inspecting the *i*-th sub-goal of `/goals/`*g* by `GET`ting the `/goals/`*g*`/`*i* entry point.

Given the references (i.e., the URLs) to a theory version and a goal-list, any client may request one or more solutions for the composed goal by `POST`ing such references to the `/solutions` entry point. The request may contain a number of *optional* parameters that correspond to the parameters shown in Tables 1 and 2, namely,

- `skip`: defines the number of solutions to be skipped (defaults to 0);
- `limit`: defines the maximum number of solutions to be retrieved (defaults to ∞);
- `within`: defines the maximum amount of time the server may spend searching for a solution (defaults to ∞);
- `every`: defines the amount of time between two consecutive solutions, meaning that a periodic solution stream is actually desired; and
- `hook`: defines the *hook* associated with the request, that is, a mnemonic name used to retrieve *ex-post* a solution previously computed by the LPaaS server.

The following optional parameters support situatedness in space:

- `where`: defines the centre of the region of validity to be considered for the solution, meaning that the resolution process should only take into account the logic theories and facts being valid within that when the resolution process starts;
- `regionSize`: defines the *size* of the aforementioned validity region; and
- `regionSpan`: defines the *variation in size* to add to the `regionSize` value at each step of the resolution process.

The `every` and `regionSpan` parameters aim to support *streams of solutions* in addition to a single set of solutions. The first initiates a time-related stream of solutions where the provided query is resolved periodically, while the second initiates a space-related stream of solutions where the provided query is resolved against a progressively widening/narrowing validity region (depending on the sign of `regionSpan`).

The aforementioned hook makes it possible to retrieve already computed solutions by `GET`ting the resource located at `/solutions/hook`, thus providing a sort of cache memory. Its value, say *h*, is of particular interest in case a client needs to consume the results of a streaming request, since it refers to the very last outcome of the resolution process. Users may then issue a `GET` request towards the `/solutions/`*h* entry point to retrieve the latter available result of solution stream, while previous solutions are accessible through an *ad-hoc* entry point.

4.2. The Development Process

Delivering an LP engine (i.e., a logic-based inference service) as a microservice in the IoT landscape means not only ensuring non-functional properties, such as modularity, loose coupling, and scalability, but also affecting the software engineering practice and thus, the development process itself. Continuous integration (CI) [50] and, most importantly, continuous delivery (CD) [51] pipelines are de-facto standards for developing microservices, since they comply with agile methodologies and effectively promote the best practices concerning software architecture and implementation.

Figure 5 shows the scheme of the LPaaS development process. The service implementation supports container-based deployment via the Docker technology, further discussed in Section 4.4, and the development process supports CI and CD to provide fast updates and improvements to devices hosting the LPaaS service.

In particular, the distributed software development [49] relies on the GitLab source code repository and Git (https://git-scm.com/) as a versioning tool. Build cycles are handled through the Gradle build system(https://gradle.org/) (migrating from Maven), while regression/integration testing amongst inner microservices exploits GitLab's CI/CD Pipelines feature (https://docs.gitlab.com/ee/ci/pipelines.html). Finally, the deployment on the Docker platform (https://www.docker.com/what-docker) relies on integration with the GitLab pipeline through the Docker Engine feature (https://docs.gitlab.com/ce/ci/docker/using_docker_build.html).

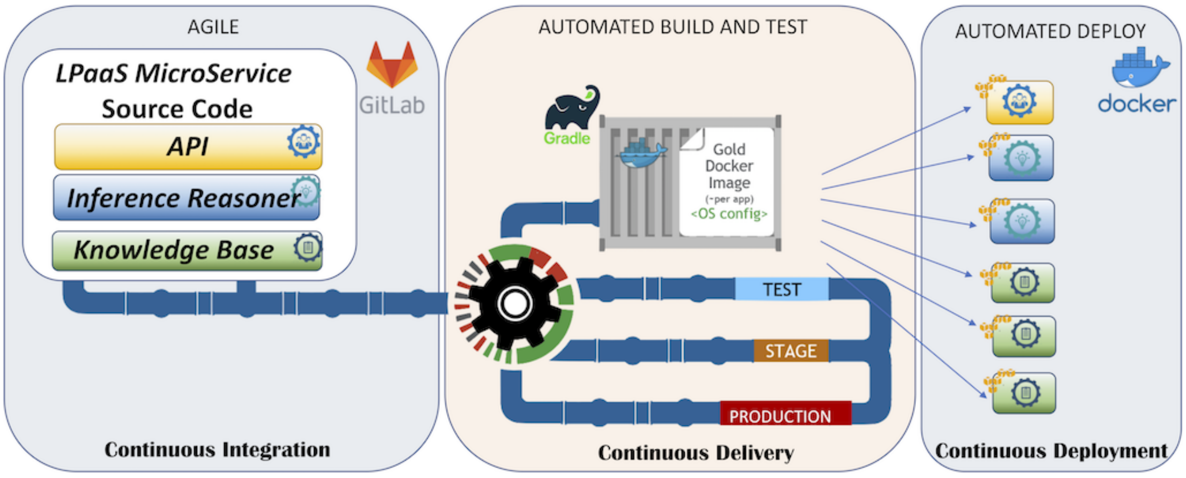


Figure 5. The LPaaS development process.

4.3. RESTful API Implementation

In order to adhere to the RESTful API description, the KB of the service is fully versioned, so any side-effects caused by the configurator or by sources either directly, through POST/PUT methods, or indirectly, via `assert/retract` predicate, on some theory automatically produces a new version of that theory. Versioned theories are referenceable by means of specific URLs and are made to be persistent through the storage layer; thanks to the uncoupling provided by the PA object, the relational specific target technology is irrelevant (http://www.oracle.com/technetwork/java/javaee/tech/persistence-jsp-140049.html). The client request always specifies (the URL of) the desired version of the theory. If no version is provided, the last available one is assumed. If no theory is provided at all, the `/theories/default` one is assumed. Analogously, for both goals and solutions entry points, if no list of goals is provided, the `/goals/default` is assumed. If no *hook* is specified, a novel and unique one is generated on the fly and provided to the client together with the solution.

Stream solutions—which require a resolution process to be re-executed periodically—are made to be resilient by making the request persistent on the storage layer. This implies that the solution stream continues to be produced even if the LPaaS server crashes and has to be restarted.

As far as security is concerned, authentication is achieved via JSON Web Tokens (JWT) technology. The LPaaS service provides—in case credentials are valid—a cryptographically signed token within the HTTP response, certifying the client's role(s). It is the client's responsibility, in order to be recognised as a configurator or a source, to provide the signed token by means of the Authorization HTTP header of any following request.

4.4. Deployment by Containerization

Containerisation is a means for developing, testing, and deploying virtually any kind of application on virtually any kind of infrastructure, either centralised or distributed. Containers consist of lightweight, portable, and scalable virtual environments wrapping a particular application—LPaaS microservices in our case—and only including the *runtime* libraries it strictly needs. Containers can be serialised, copied on different machines, and then re-executed or arbitrarily replicated, regardless of the specific configuration of the hosting machine. Docker (https://www.docker.com) is arguably the most interoperable technology supporting containerisation on mainstream operating systems, such as Windows, Linux, or MacOS. Deploying an application with Docker requires to build an *image* of the application, that is, an executable package featuring everything needed to run the application—the code, a runtime, libraries, environment variables, and configuration files.

In our case, a Dockerfile is created for the LPaaS image using the Docker runtime. Then, a containerised application can be deployed either as a standalone *service*, or as a *microservice* that is part of a *stack* of services depending on one another.

The business logic component of the LPaaS architecture is executable as a standalone application as well as the data layer component, whichm at presentm may be based on either an embedded database provided by the Payara Micro edition (https://www.payara.fish/payara_micro) or an external DB such as PostgreSQL (https://www.postgresql.org/) thanks to JPA. Actually, the LPaaS engine and the KB are independent microservices, deployed on a common *network*, isolated from any other stack of services. Such a stack can then be run either on a single machine or on a *swarm* of containers/hosts through the *Docker Swarm* functionality, thus becoming fully distributed. The LPaaS deployment described in this paper follows the latter approach, taking advantage of all the aforementioned functionalities provided by the Docker technology.

5. A Conceptual Use Case

The design of terms of microservices paves the way to the use of LP inference techniques as *micro-intelligence sources* to meet the requirements of light, efficient, scalable, and distributed intelligence in IoT scenarios. This is especially effective when mixed with other non-symbolic approaches. In this section, we envision a use case in the domain of energy grid distribution, aimed at demonstrating the potential of such an approach for engineering intelligence in distributed systems. The system we envision (Figure 6) is therefore composed of thousands or millions of users and their own generators, becoming producers and consumers of electricity [52], interconnected through an electricity network that is supposed to be autonomous and intelligent.

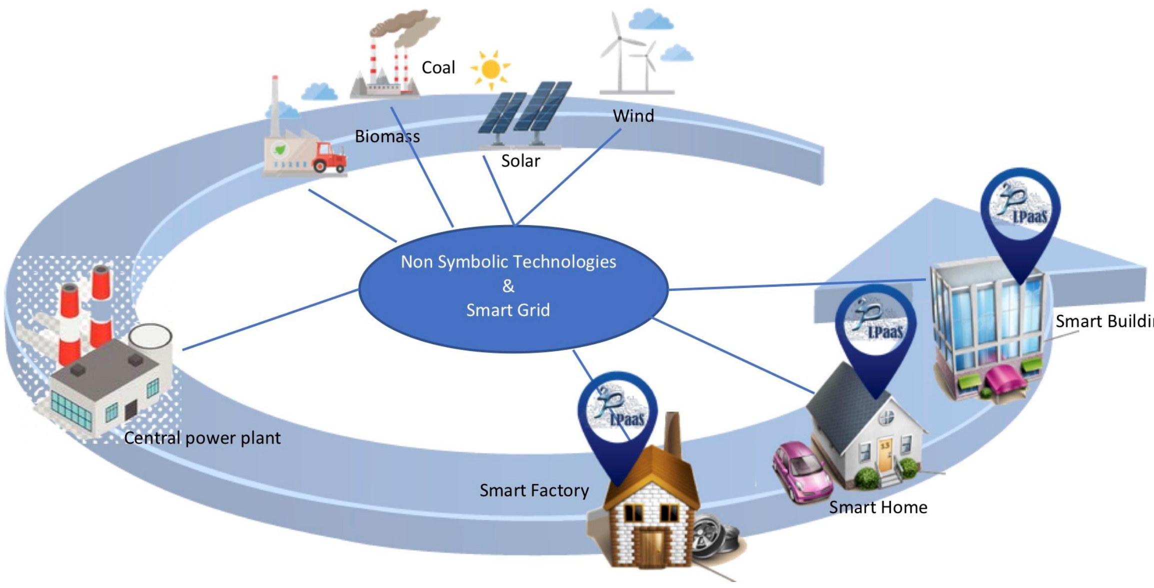


Figure 6. The LPaaS smart energy application scenario.

5.1. The State of Art

Figure 6 depicts a typical scenario in modern "smart" energy grid distribution solutions. Energy harvesting farms, either featuring renewable sources or not, are connected to a power plant in charge of distribution of the generated power to many heterogeneous (i.e., in scale and goals) destinations, such as industries, public places, and private houses.

What is "smart" in a grid is its ability to timely and correctly deal with fluctuations in offer and demand while keeping power distribution within safety and fairness boundaries set, i.e., by local administration, with little or no human intervention—that is, autonomously. To do so, the most widespread approach nowadays is to rely on non-symbolic approaches, for instance, exploiting ML techniques to learn to predict offer and demand based on historical production/consumption data or statistical analyses to learn correlations between data so as to automatically generate rules and policies about energy distribution [53]. For instance, reference [54] explored automatically creating site-specific prediction models for solar power generation from National Weather Service (NWS) weather forecasts using ML techniques.

In such a scenario, many challenges need to be dealt with in order to achieve a better integration of flexible demand (demand response and demand side management) with distributed generation [55,56]. For instance, the system has to be designed and operate as an integrated unit, a far from trivial issue when ownership, decision-making, and operation are distributed. Here, energy management can be implemented in a global and centralised fashion by a single entity, e.g., a cloud-based grid management application, or in a local and decentralised manner by means of multiple components which are able to adjust a site-specific configuration while converging towards system-level goals and in compliance with system-level safety boundaries.

Smart grids for energy management are meant to enable a future power market with appropriate communication and intelligent energy control technology that is able to smartly maintain a balance between the distributed energy resources and the demand requirement in a user-friendly manner. In particular, according to reference [57–59], a decentralised micro-grid represents a key tool to improve the energy demand and supply management in the smart grid. This is achieved by exploiting information about electricity consumption, transmission configuration, and advanced technology for harvesting renewable energy on a finer demand/supply scale. The grid constantly monitors the demand for electricity and dispatches generators to satisfy demand as it rises and falls. Since electricity demand is highly predictable when aggregating over thousands of buildings and homes, today's grid is able to accurately plan in advance which generators to dispatch and when, to satisfy demand. Moreover, *a-priori* knowledge about the energy load pattern can help to reshape the load and cut the energy demand curve, thus allowing better management and distribution of the energy in smart grid energy systems.

The next section builds on the scenario just described to emphasise how the LPaaS symbolic approach to distributed intelligence in IoT-like applications can effectively complement non-symbolic approaches.

5.2. The LPaaS Contribution

Within the application domain depicted above, LPaaS could help by mitigating some of the issues experienced by non-symbolic approaches, as well as by enabling novel forms of distributed and local reasoning, adding all the symbolic techniques advantages to the system. In particular,

1. When deploying a system for the first time, either a ML model has been trained on artificially produced synthetic data (or on similar data coming from a similar environment as the one target of the deployment) or it will inevitably suffer from the so-called "cold start" problem—the system is initially unusable and requires a training phase and then starts making increasingly accurate predictions as soon as it gathers training data from the deployment environment.
2. ML or statistical approaches get better as more data is available for training and online learning—thus, for instance, they are better applied on energy profiles of many private houses clustered together based on similarity, rather than at the level of a single house.
3. Stemming from point §2, non-symbolic approaches, in general, are good at generalising models by abstracting away from specific peculiarities and focussing on the commonalities between the different data samples. Consequently, they are usually deficient when tailoring models and producing subsequent predictions of the specific traits of a single data sample.
4. Fixing models in case their accuracy starts worsening, for instance, because the conditions of the deployment environment have abruptly changed—i.e., a neighbourhood changing power generators all at once—is extremely difficult, if not impossible. On the one hand, since non-symbolic approaches rely on data samples to actually learn to generalise, they are intrinsically slow to adapt to changes and unforeseen situations. On the other hand, their lack of transparency and explainability hinders data scientists acquiring a complete understanding of what is going wrong .

The first issue described above can be mitigated by a symbolic approach, such as the one proposed by LPaaS, in that white-box models are built upon domain experts' knowledge rather than out of data processing. As such, they may fail to model hidden patterns buried deep into the data, but at least are immediately usable and guaranteed to comply with the designed goals and requirements. As far as issues §2 and §3 are concerned, LPaaS is precisely meant to model and enable seamless access to situated knowledge and inference services, thus making it possible to take into account the specific features of each knowledge domain. In the smart energy grid scenario, for instance, it is almost impossible for a non-symbolic (either statistical or ML) approach to account for the so-called "outliers", such as a single house out of a neighbourhood behaving differently from the rest. Tts specificities would be abstracted away by the general model learnt. Instead, by complementing ML with LPaaS, for instance, it would be possible to override the inference rules learnt by non-symbolic approaches from the global data gathered in the neighbourhood with house-specific rules designed for the specific purpose. Finally, as far as issue §4 is concerned, the white-box models produced by symbolic approaches, including LPaaS, are inherently easier to fix. The inference process is, in fact, transparent and capable of explaining outcomes, so that tracking errors and causally connecting the output to the input is easier. More generally, changing rules, policies, and facts in the knowledge base involved in the inference process is easier especially with approaches based on logic programming, such as LPaaS, thanks to meta-programming features enabling developers to program how the inference engine should modify itself at run-time and when—that is, as a consequence of which events.

In the energy grid scenario described in previous sub-section and depicted in Figure 6, the responsibility for global and local energy management policies could be split as follows: globally, non-symbolic approaches may take advantage of the scale to train clustering and classification models regarding, for instance, consumption/production profiles of groups of customers and power plants, and fluctuations of offer and demand at the scale of neighbourhoods, districts, or whole cities. Instead, locally, each consumer unit (house, office, industry, ...) could host one or more LPaaS services, either for the unit as a whole or for each device of interest, in charge of *(i)* communicating with the smart grid platform any potentially relevant data about energy consumption and production, and *(ii)* locally enforcing safety/fairness rules and/or management policies, possibly over-riding the requests globally computed by the smart grid platform. As an example, Figure 7 considers the case of energy management in a Smart House. Each smart device is equipped with an LPaaS service which makes decisions about the local situation of the device (for instance, depending on current energy usage, whether to turn air conditioning on, whether to store energy produced in excess for later or dispatch it to the grid, etc.) and communicating energy consumption/production related information to the smart grid when requested. There, the Energy Management Manager can collect all information about the house in order to schedule the devices accordingly to users' policies and energy available at that moment.

In such a scenario, based on existing literature, it may happen that

- Non-symbolic techniques—such as support vector machines, multiple regression schemas, and different classification algorithms—are exploited to predict energy generation from renewable sources, such as solar panels and wind turbines, based on data coming from the National Weather Service (NWS) weather forecasts for specific sites [54]. Doing so incorporates site-specific characteristics into the models, such as shade coming from nearby trees, average wind strength, etc., which should be fine-tuned when transferring the application of the model to another domain. If a symbolic approach, such as LPaaS, is exploited in synergy, instead, these peculiarities can be modelled separately on a local-only basis as a white-box model, and they can be used to modulate or over-ride the predictions coming from the smart grid. For instance, there could be a simple rule over-riding the estimated solar prediction if the weather is bad despite positive weather forecasts. Additional benefits would be that *(i)* changing such over-ride rules on the fly is possible thanks to logic programming meta-programming features, and *(ii)* local decision-making is easily inspectable and accountable since inference rules stem from a white-box model.
- In reference [60], energy load forecasting based on Deep Neural Networks—specifically, Long Short Term Memory (LSTM) algorithms—was used to make predictions about future energy demand, at both aggregate and individual site levels. Good results were shown for an hourly resolution of the prediction horizon. Based on these predictions, the energy production was scheduled. LPaaS could also help also in regard to events that are not considered in the training dataset, such as a strike or holiday closing. Yet again, local rules crafted upon need by domain experts may over-ride the forecasts dispatched by the smart grid, also providing the same added benefits mentioned above.

It is worth remarking here that on the one hand, LPaaS is complementary to non-symbolic approaches, as described above, but on the other hand, ML is complementary to the symbolic approach of LPaaS as well. In fact, it is very unlikely for LPaaS or similar techniques to outperform non-symbolic approaches in *(i)* handling large amounts of data, *(ii)* coping with incomplete and uncertain information, and *(iii)* automatically learning models without any a-priori knowledge. Hence, our position to exploit them in synergy rather than separately.

To the best of our knowledge, symbolic and non-symbolic techniques are integrated manually. In particular, symbolic rules rely on manually-crafted expert knowledge. A possible research perspective here could consider the automatic synthesis of rules based on some suitable inference process; however, such an interesting investigation is outside the scope of this paper.

Also, we would like to emphasise that all of the above is made possible by the re-interpretation of distributed logic programming provided by the LPaaS model and technology, which, above all, is meant to promote and support the distribution of micro-intelligence wherever needed. A similar scenario would not be possible with traditional logic programming, where there is no concept of situatedness in space and time, and where only a single monolithic theory of the whole world is considered. Related works, sources of inspiration for the re-interpretation of logic programming, and for the design of LPaaS are discussed in next section.

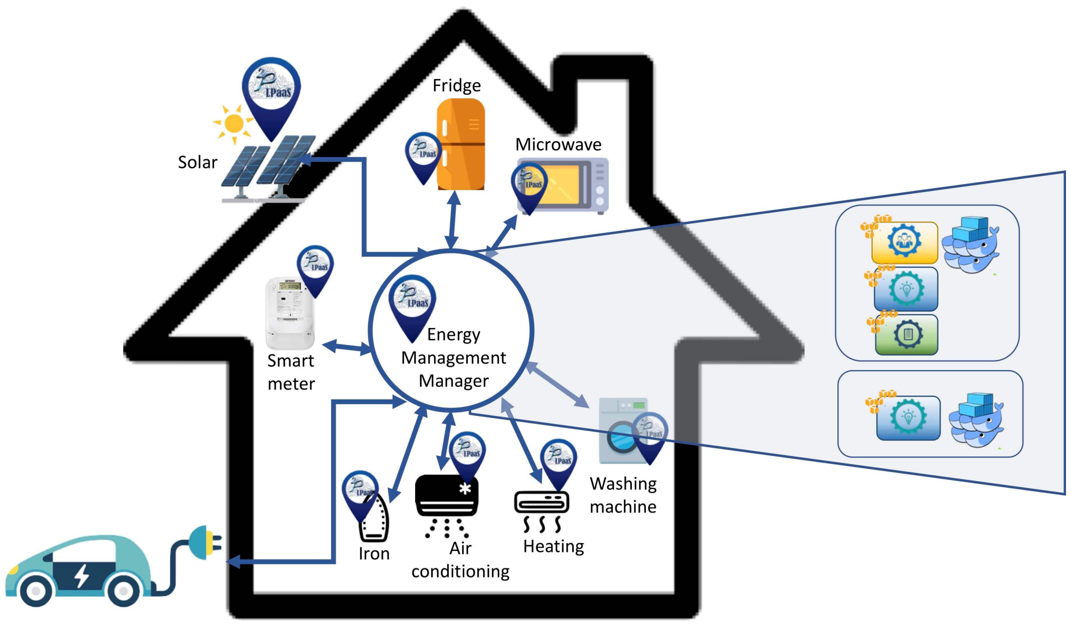


Figure 7. The LPaaS energy management in the smart house.

6. Related Work

In order to optimally support the wide variety of IoT applications and user needs, intelligence is needed in both the smart objects populating the scene and the infrastructure on which they rely for communication and coordination [61]. Here, intelligence is not only related to the processing of data, but also to security, quality of service, network configuration, etc. [23,61]. Moreover, there is no single place where intelligence can be placed or activated. Depending on the situation it may be better to have it directly on-board devices, i.e., for efficiency and responsiveness, or centralised in the cloud, i.e., to have ML pipelines operating on all available data.

To this end, and exploiting symbolic reasoning, reference [62] discusses a comprehensive logic programming framework designed to support intelligent composition of Web services. The underlying model relies on representing web services as actions, each described by a logic programming theory. This promotes the use of a logic-based approach and symbolic reasoning to address some IoT challenges, in particular, making a step towards interoperability and composeability. Symbolic reasoning capabilities and features are also discussed in reference [63], highlighting the importance of high-level programming on resource-constrained devices. There, they develop a lightweight middleware to support distributed intelligence exploiting event detection that combines fuzzy reasoning and additional symbolic knowledge—in the form of data, rules, and executable code—to enrich the semantics of events. With respect to both of these works, we adopted a similar logic programming approach, offering high-level programming and symbolic reasoning capabilities.

Nevertheless, LPaaS goes beyond this w.r.t. context-awareness and situatedness, by supporting the injection of intelligence within existing services/agents via either a network-reachable API or direct embedding of the LPaaS service.

On the side of architectures and middleware for scalable and ubiquitous computing in IoT, many research works promote the usage of event-driven SOA (EDA-SOA) [64–66], mostly due to their effectiveness in terms of scalability, interoperability, reliability, and responsiveness. In particular, reference [64] shows how challenges like scalability can be addressed by exploiting SOAP-based, RESTful, and broker-based architectures. LPaaS itself follows the EDA-SOA principles in regard to the model and architecture design. In particular, the proposed notion of a stream of solutions was inspired by the EDA principles, in that it allows a "signal" containing the new solution to be sent. By following the EDA-SOA principles, is possible to achieve distribution, interoperability, and scalability as confirmed by the abovementioned research works.

Generally speaking, as highlighted in reference [61,67], distributed intelligence can contribute to increased interoperability as well as better usage of scarce resources (enabling local reasoning), and this is exactly the aim of LPaaS.

Moreover, the logic-based approach brings advantages related to symbolic reasoning, such as provable properties, sound and explainable inference, and the fact that the intelligent behaviour exhibited by the system becomes explainable—converse to what happens with the majority of non-symbolic approaches. As a step further, references [67,68] show the importance of combining symbolic and non-symbolic techniques to achieve intelligent behaviours. Hybrid intelligent architectures [69] synergically combine the strengths of diverse computational intelligence paradigms and exploit both the domain knowledge and the training data to solve difficult learning tasks. To this end, LPaaS, thanks to its light-weight architecture, could be easily combined with other techniques in order to achieve hybrid reasoning—where symbolic and non-symbolic techniques fruitfully combine to produce intelligent behaviour.

7. Materials and Methods

The LPaaS prototype is freely available on Bitbucket [70] with the corresponding installation guide. It was built on the top of the tuProlog system, a lightweight Prolog engine which provides the required interoperability and customisation capabilities [71].

A prototype implementation of LPaaS as a *RESTful Web Service* was provided in accordance with the REST API discussed above, showing the effectiveness of LPaaS in supporting REST and distributing situated intelligence. A full discussion on the case studies implemented exploiting the LPaaS RESTful Web Service is presented in reference [19].

8. Conclusions

IoT scenarios are increasingly demanding intelligence to be scattered everywhere and to support adaptation and self-management at different degrees. Software engineering challenges in this context concern interoperability, efficiency, and scalability, but also a careful design of both the ML pipeline and the logic inference engine, separately as well as in complementary exploitation. Such issues do not merely concern the application or service logic and runtime, but also impact on the software development process.

The LPaaS model and architecture, discussed in depth in this work, aims to fit such a challenging context by introducing an LP microservice as a provider of symbolic reasoning techniques in IoT scenarios, complementing non-symbolic ones. Accordingly, we discussed an envisioned use case regarding energy grid operations to highlight the benefits of the LPaaS approach in terms of scalability, interoperability, and reactivity, but also in terms of observability, explainability, and accountability, fostering the idea that symbolic approaches have the potential to be key players in the future of large-scale intelligent systems.

Open issues in this research thread are mostly concerned with how to integrate the two complementary—no longer antithetic—approaches, possibly in an automated way. As relatively recent examples, three techniques have been presented by the same research group in the context of adaptation and scaling of transactional memories: divide and conquer [72], bootstrapping [73], and boosting [74]. In the first, black-box modelling (thus, non-symbolic approaches) and white-box modelling (that is, symbolic approaches) are used separately on different sub-problems, and their output is reconciled afterwards in a single comprehensive objective function. In the second, white-box models are hand-crafted with the purpose of providing initial, synthetic training data to black-box ones to be later refined with real data (in the form of online training). In the third, a ML pipeline made up of several black-box models in a cascade is used to incrementally reduce the error of a white-box model—by learning the error of an objective function instead of the function itself.

The extent to which a LP-based approach to symbolic reasoning can fit the above techniques is yet unexplored and will be subject of further work.

Author Contributions: All authors contributed to the intellectual content of this paper following these requirements: (1) significant contributions to the conception and design, acquisition of data or analysis and interpretation of data; (2) drafting or revising the article for intellectual content; and (3) final approval of the published article. In particular: Conceptualization, A.O., E.D., R.C. and S.M.; Methodology, R.C. and S.M.; Software, R.C. and G.C.; Validation, A.O. and E.D.; Data Curation, R.C. and G.C.; Writing Original Draft Preparation, R.C.; Writing Review & Editing, S.M., G.C., E.D. and A.O.; Supervision, A.O.

Funding: This research received no external funding.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Larrucea, X.; Combelles, A.; Favaro, J.; Taneja, K. Software Engineering for the Internet of Things. *IEEE Softw.* **2017**, *34*, 24–28. [CrossRef]
2. Lippi, M.; Mamei, M.; Mariani, S.; Zambonelli, F. Coordinating Distributed Speaking Objects. In Proceedings of the IEEE 37th International Conference on Distributed Computing Systems (ICDCS 2017), Atlanta, GA, USA, 5–8 June 2017; pp. 1949–1960. [CrossRef]
3. Arsénio, A.; Serra, H.; Francisco, R.; Nabais, F.; Andrade, J.; Serrano, E. Internet of Intelligent Things: Bringing Artificial Intelligence into Things and Communication Networks. In *Inter-Cooperative Collective Intelligence: Techniques and Applications*; Xhafa, F., Bessis, N., Eds.; Studies in Computational Intelligence; Springer: Berlin/Heidelberg, Germany, 2014; Volume 495, pp. 1–37. [CrossRef]
4. Fortino, G.; Rovella, A.; Russo, W.; Savaglio, C. On the Classification of Cyberphysical Smart Objects in the Internet of Things. In Proceedings of the CEUR Workshop on UBICITEC-2014—Networks of Cooperating Objects for Smart Cities 2014, Berlin, Germany, 4 April 2014; Volume 1156, pp. 86–94.
5. Muggleton, S.H.; Schmid, U.; Zeller, C.; Tamaddoni-Nezhad, A.; Besold, T. Ultra-Strong Machine Learning: comprehensibility of programs learned with ILP. *Mach. Learn.* **2018**, *107*, 1119–1140. [CrossRef]
6. Besold, T.R.; Garcez, A.D.; Stenning, K.; van der Torre, L.; van Lambalgen, M. Reasoning in Non-probabilistic Uncertainty: Logic Programming and Neural-Symbolic Computing as Examples. *Minds Mach.* **2017**, *27*, 37–77. [CrossRef]
7. Brooks, R.A. Intelligence Without Reason. In *Proceedings of the 12th International Joint Conference on Artificial Intelligence (IJCAI 1991)*; Mylopoulos, J., Reiter, R., Eds.; Morgan Kaufmann Publishers Inc.: San Francisco, CA, USA, 1991; Volume 1, pp. 569–595.
8. Brooks, R.A. Intelligence without Representation. *Artif. Intell.* **1991**, *47*, 139–159. [CrossRef]
9. LeCun, Y.; Bengio, Y.; Hinton, G. Deep learning. *Nature* **2015**, *521*, 436–444. [CrossRef] [PubMed]
10. Chen, X.W.; Lin, X. Big Data Deep Learning: Challenges and Perspectives. *IEEE Access* **2014**, *2*, 514–525. [CrossRef]
11. Najafabadi, M.M.; Villanustre, F.; Khoshgoftaar, T.M.; Seliya, N.; Wald, R.; Muharemagic, E. Deep learning applications and challenges in big data analytics. *J. Big Data* **2015**, *2*, 1. [CrossRef]

12. Association for Computing Machinery US Public Policy Council (USACM). Statement on Algorithmic Transparency and Accountability. 2017. Available online: https://www.acm.org/binaries/content/assets/public-policy/2017_usacm_statement_algorithms.pdf (accessed on 1 July 2018).
13. EU Commission. Algorithmic Awareness-Building. 2018. Available online: https://ec.europa.eu/digital-single-market/en/algorithmic-awareness-building (accessed on 1 July 2018).
14. Dix, A. Human–computer interaction, foundations and new paradigms. *J. Vis. Lang. Comput.* **2017**, *42*, 122–134. [CrossRef]
15. Garnelo, M.; Arulkumaran, K.; Shanahan, M. Towards deep symbolic reinforcement learning. In Proceedings of the Neural Information Processing Systems (NIPS) 2016—Workshop on Deep Reinforcement Learning, Barcelona, Spain, 5–10 December 2016.
16. Marcus, G. Deep Learning: A Critical Appraisal. *ArXiv* **2018**, arxiv:1801.00631.
17. Hatcher, W.G.; Yu, W. A Survey of Deep Learning: Platforms, Applications and Emerging Research Trends. *IEEE Access* **2018**, *6*, 24411–24432. [CrossRef]
18. Fortino, G.; Russo, W.; Savaglio, C.; Shen, W.; Zhou, M. Agent-Oriented Cooperative Smart Objects: From IoT System Design to Implementation. *IEEE Trans. Syst. Man Cybern. Syst.* **2017**, 1–18. [CrossRef]
19. Calegari, R.; Denti, E.; Mariani, S.; Omicini, A. Logic Programming as a Service (LPaaS): Intelligence for the IoT. In Proceedings of the 2017 IEEE 14th International Conference on Networking, Sensing and Control (ICNSC 2017), Calabria, Italy, 16–18 May 2017; pp. 72–77. [CrossRef]
20. Calegari, R.; Denti, E.; Mariani, S.; Omicini, A. Logic Programming as a Service. *Theory Pract. Logic Program.* **2018**, *18*, 1–28. [CrossRef]
21. Calegari, R.; Ciatto, G.; Mariani, S.; Denti, E.; Omicini, A. Micro-intelligence for the IoT: SE Challenges and Practice in LPaaS. In Proceedings of the 2018 IEEE International Conference on Cloud Engineering (IC2E 2018), IEEE Computer Society, Orlando, FL, USA, 17–20 April 2018; pp. 292–297. [CrossRef]
22. Gubbi, J.; Buyya, R.; Marusic, S.; Palaniswami, M. Internet of Things (IoT): A vision, architectural elements, and future directions. *Future Gener. Comput. Syst.* **2013**, *29*, 1645–1660. [CrossRef]
23. Botta, A.; De Donato, W.; Persico, V.; Pescapé, A. On the Integration of Cloud Computing and Internet of Things. In Proceedings of the 2014 International Conference on Future Internet of Things and Cloud (FiCloud 2014), Barcelona, Spain, 27–29 August 2014; pp. 23–30. [CrossRef]
24. Perera, C.; Zaslavsky, A.; Christen, P.; Georgakopoulos, D. Context Aware Computing for The Internet of Things: A Survey. *IEEE Commun. Surv. Tutor.* **2014**, *16*, 414–454. [CrossRef]
25. Zhou, L.; Pan, S.; Wang, J.; Vasilakos, A.V. Machine learning on big data: Opportunities and challenges. *Neurocomputing* **2017**, *237*, 350–361. [CrossRef]
26. Agerri, R.; Bermudez, J.; Rigau, G. IXA pipeline: Efficient and Ready to Use Multilingual NLP tools. In Proceedings of the 9th Language Resources and Evaluation Conference (LREC 2014), Reykjavik, Iceland, 26–31 May 2014; pp. 3823–3828.
27. Bologna, G.; Hayashi, Y. A Rule Extraction Study from SVM on Sentiment Analysis. *Big Data Cogn. Comput.* **2018**, *2*, 6. [CrossRef]
28. Hoehndorf, R.; Queralt-Rosinach, N. Data science and symbolic AI: Synergies, challenges and opportunities. *Data Sci.* **2017**, *1*, 27–38. [CrossRef]
29. Consortium, G.O. The Gene Ontology (GO) database and informatics resource. *Nucleic Acids Res.* **2004**, *32*, D258–D261. [CrossRef] [PubMed]
30. Brownlee, J. *Clever Algorithms: Nature-Inspired Programming Recipes*; Lulu Press: Morrisville, NC, USA, 2011.
31. Palù, A.D.; Torroni, P. 25 Years of Applications of Logic Programming in Italy. In *A 25-Year Perspective on Logic Programming*; Dovier, A., Pontelli, E., Eds.; Springer: Berlin/Heidelberg, Germany, 2010; pp. 300–328. [CrossRef]
32. Veanes, M.; Hooimeijer, P.; Livshits, B.; Molnar, D.; Bjorner, N. Symbolic Finite State Transducers: Algorithms and Applications. *ACM SIGPLAN Not.* **2012**, *47*, 137–150. [CrossRef]
33. Belta, C.; Bicchi, A.; Egerstedt, M.; Frazzoli, E.; Klavins, E.; Pappas, G.J. Symbolic planning and control of robot motion [Grand Challenges of Robotics]. *IEEE Robot. Autom. Mag.* **2007**, *14*, 61–70. [CrossRef]
34. Martelli, M. Constraint logic programming: Theory and applications. In *1985–1995: Ten Years of Logic Programming in Italy*; Sessa, M., Ed.; Palladio Editrice: Salerno, Italy, 1995; pp. 137–166.

35. Rosenberg, D.; Boehm, B.; Wang, B.; Qi, K. Rapid, Evolutionary, Reliable, Scalable System and Software Development: The Resilient Agile Process. In Proceedings of the 2017 International Conference on Software and System Process (ICSSP 2017), Paris, France, 5–7 July 2017; pp. 60–69. [CrossRef]
36. Familiar, B. *Microservices, IoT, and Azure: Leveraging DevOps and Microservice Architecture to Deliver SaaS Solutions*, 1st ed.; Apress: Berkely, CA, USA, 2015.
37. Erl, T. *Service-Oriented Architecture: Concepts, Technology, and Design*; Prentice Hall/Pearson Education International: Upper Saddle River, NJ, USA , 2005.
38. Rahman, H.; Rahmani, R. Enabling distributed intelligence assisted Future Internet of Things Controller (FITC). *Appl. Comput. Inform.* **2018**, *14*, 73–87. [CrossRef]
39. Bonomi, F.; Milito, R.; Zhu, J.; Addepalli, S. Fog Computing and Its Role in the Internet of Things. In Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing (MCC 2012), Helsinki, Finland, 17 August 2012; pp. 13–16. [CrossRef]
40. Calegari, R.; Denti, E.; Dovier, A.; Omicini, A. Extending Logic Programming with Labelled Variables: Model and Semantics. *Fundam. Inform.* **2018**, *161*, 53–74. [CrossRef]
41. Calegari, R.; Denti, E.; Dovier, A.; Omicini, A. Labelled Variables in Logic Programming: Foundations. In Proceedings of the CILC 2016–Italian Conference on Computational Logic, Milano, Italy, 20–22 June 2016; CEUR-WS: Milano, Italy, 2016; Volume 1645, pp. 5–20.
42. Robinson, J.A. A Machine-Oriented Logic Based on the Resolution Principle. *J. ACM* **1965**, *12*, 23–41. [CrossRef]
43. De, S.; Barnaghi, P.; Bauer, M.; Meissner, S. Service modelling for the Internet of Things. In Proceedings of the Federated Conference on Computer Science and Information Systems (FedCSIS 2011), Szczecin, Poland, 18–21 September 2011; pp. 949–955.
44. Calegari, R.; Ciatto, G.; Mariani, S.; Denti, E.; Omicini, A. Logic Programming in Space-Time: The Case of Situatedness in LPaaS. In Proceedings of the WOA 2018–19th Workshop "From Objects to Agents", Palermo, Italy, 28–29 June 2018; in press.
45. Deransart, P.; Dbali, A.E.; Cervoni, L. *Prolog: The Standard. Reference Manual*; Springer: New York, NY, USA, 1996. [CrossRef]
46. Beierle, C.; Hedtstück, U.; Pletat, U.; Schmitt, P.; Siekmann, J. An order-sorted logic for knowledge representation systems. *Artif. Intell.* **1992**, *55*, 149–191. [CrossRef]
47. Calegari, R.; Denti, E.; Mariani, S.; Omicini, A. Logic Programming as a Service in Multi-Agent Systems for the Internet of Things. *Int. J. Grid Util. Comput.* **2018**, in press.
48. Fielding, R.T. Architectural Styles and the Design of Network-based Software Architectures. PhD Thesis, University of California, Irvine, CA, USA, 2000.
49. LPaaS. Home Page. Available online: http://lpaas.apice.unibo.it/ (accessed on 1 July 2018).
50. Humble, J.; Farley, D. *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*; Addison-Wesley/Pearson Education: Boston, MA, USA, 2010.
51. Duvall, P.M.; Matyas, S.; Glover, A. *Continuous Integration: Improving Software Quality and Reducing Risk*; Addison-Wesley/Pearson Education: Boston, MA, USA, 2007.
52. Gómez, M.; Cámara, M.Á.; Jiménez, E.; Martínez-Cámara, E. A new energetic scenario with renewable energy. In Proceedings of the International Conference on Renewable Energies and Power Quality (ICREPQ 2010), Granada, Spain, 23–25 March 2010.
53. Fallah, S.N.; Deo, R.C.; Shojafar, M.; Conti, M.; Shamshirband, S. Computational Intelligence Approaches for Energy Load Forecasting in Smart Energy Management Grids: State of the Art, Future Challenges, and Research Directions. *Energies* **2018**, *11*, 596. [CrossRef]
54. Sharma, N.; Sharma, P.; Irwin, D.; Shenoy, P. Predicting solar generation from weather forecasts using machine learning. In Proceedings of the IEEE International Conference on Smart Grid Communications (SmartGridComm 2011), Brussels, Belgium, 17–20 October 2011; pp. 528–533. [CrossRef]
55. Clastres, C. Smart grids: Another step towards competition, energy security and climate change objectives. *Energy Policy* **2011**, *39*, 5399–5408. [CrossRef]

56. Yu, X.; Cecati, C.; Dillon, T.; Simões, G.M. The New Frontier of Smart Grids. *IEEE Ind. Electron. Mag.* **2011**, *5*, 49–63. [CrossRef]
57. Vonk, B.M.J.; Nguyen, P.H.; Grand, M.O.W.; Slootweg, J.G.; Kling, W.L. Improving Short-term load forecasting for a local energy storage system. In Proceedings of the 47th International Universities Power Engineering Conference (UPEC 2012), London, UK, 4–7 September 2012. [CrossRef]
58. Wijaya, T.K.; Eberle, J.; Aberer, K. Symbolic Representation of Smart Meter Data. In Proceedings of the Joint EDBT/ICDT 2013 Workshops (EDBT 2013), Genoa, Italy, 18–22 March 2013; pp. 242–248. [CrossRef]
59. Hernández, L.; Baladrón, C.; Aguiar, J.M.; Carro, B.; Sánchez-Esguevillas, A.; Lloret, J. Artificial neural networks for short-term load forecasting in microgrids environment. *Energy* **2014**, *75*, 252–264. [CrossRef]
60. Marino, D.L.; Amarasinghe, K.; Manic, M. Building energy load forecasting using Deep Neural Networks. In Proceedings of the 42nd Annual Conference of the IEEE Industrial Electronics Society (IECON 2016), Florence, Italy, 23–26 October 2016; pp. 7046–7051. [CrossRef]
61. Van den Abeele, F.; Hoebeke, J.; Teklemariam, G.K.; Moerman, I.; Demeester, P. Sensor Function Virtualization to Support Distributed Intelligence in the Internet of Things. *Wirel. Pers. Commun.* **2015**, *81*, 1415–1436. [CrossRef]
62. Pontelli, E.; Cao Son, T.; Baral, C. A Logic Programming Based Framework for Intelligent Web Service Composition. In *Managing Web Service Quality: Measuring Outcomes and Effectiveness*; Khaled, M.K., Ed.; IGI Global: Hershey, PA, USA, 2009; pp. 193–221. [CrossRef]
63. Gaglio, S.; Lo Re, G.; Martorella, G.; Peri, D. High-Level Programming and Symbolic Reasoning on IoT Resource Constrained Devices. In *Internet of Things. User–Centric IoT*; Giaffreda, R., Vieriu, R.L., Pasher, E., Bendersky, G., Jara, A.J., Rodrigues, J.J., Dekel, E., Mandler, B., Eds.; Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering (LNICST); Springer: Cham, Switzerland, 2015; Volume 150, pp. 58–63. [CrossRef]
64. Gupta, P.; Mokal, T.P.; Shah, D.D.; Satyanarayana, K.V.V. Event-Driven SOA-Based IoT Architecture. In *International Conference on Intelligent Computing and Applications*; Dash, S.S., Das, S., Panigrahi, B.K., Eds.; Advances in Intelligent Systems and Computing (AISC); Springer: Singapore, 2018; Volume 632, pp. 247–258. [CrossRef]
65. Guerrero-Contreras, G.; Navarro-Galindo, J.L.; Samos, J.; Garrido, J.L. A collaborative semantic annotation system in health: Towards a SOA design for knowledge sharing in ambient intelligence. *Mob. Inf. Syst.* **2017**, *2017*, 4759572. [CrossRef]
66. Malekzadeh, B. Event-Driven Architecture and SOA in Collaboration-A sTudy of How Event-Driven Architecture (EDA) Interacts and Functions Within Service-Oriented Architecture (SOA). Master's Thesis, University of Gothenburg, Gothenburg, Sweden, 2010.
67. Zarri, G.P. High-Level Knowledge Representation and Reasoning in a Cognitive IoT/WoT Context. In *Cognitive Computing for Big Data Systems Over IoT: Frameworks, Tools and Applications*; Sangaiah, A.K., Thangavelu, A., Meenakshi Sundaram, V., Eds.; Lecture Notes on Data Engineering and Communications Technologies (LNDECT); Springer: Cham, Switzerland, 2018; Volume 14, pp. 223–262. [CrossRef]
68. Ghosh, J.; Taha, I. A neuro-symbolic hybrid intelligent architecture with applications. In *Recent Advances in Artificial Neural Networks*; Jain, L.C., Fanelli, A.M., Eds.; CRC Press: Boca Raton, FL, USA, 2000; pp. 2–37.
69. McGarry, K.; Wermter, S.; MacIntyre, J. Hybrid neural systems: From simple coupling to fully integrated neural networks. *Neural Comput. Surv.* **1999**, *2*, 62–93.
70. LPaaS tuProlog. Home Page. 2017. Available online: https://bitbucket.org/tuProlog/lpaas-tuprolog/ (accessed on 1 July 2018).
71. Denti, E.; Omicini, A.; Ricci, A. tuProlog: A Light-weight Prolog for Internet Applications and Infrastructures. In Proceedings of the Practical Aspects of Declarative Languages, 3rd International Symposium (PADL 2001), Las Vegas, NV, USA, 11–12 March 2001; Ramakrishnan, I.V., Ed.; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2001; Volume 1990, pp. 184–198. [CrossRef]
72. Didona, D.; Romano, P.; Peluso, S.; Quaglia, F. Transactional Auto Scaler: Elastic Scaling of Replicated In-Memory Transactional Data Grids. *ACM Trans. Auton. Adapt. Syst.* **2014**, *9*, 1–32. [CrossRef]

73. Duarte, F.; Gil, R.; Romano, P.; Lopes, A.; Rodrigues, L. Learning Non-deterministic Impact Models for Adaptation. In Proceedings of the 13th International Conference on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2018), Gothenburg, Sweden, 28–29 May 2018; ACM: New York, NY, USA, 2018; pp. 196–205. [CrossRef]
74. Didona, D.; Felber, P.; Harmanci, D.; Romano, P.; Schenker, J. Identifying the optimal level of parallelism in transactional memory applications. *Computing* **2015**, *97*, 939–959. [CrossRef]

© 2018 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (http://creativecommons.org/licenses/by/4.0/).