The Journal of Systems and Software 213 (2024) 112041

Contents lists available at ScienceDirect

The Journal of Systems & Software

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

Modeling microservice architectures ☆

Javier Esparza-Peidro *, Francesc D. Muñoz-Escoí, José M. Bernabéu-Aubán

Instituto Universitario Mixto Tecnológico de Informática; Universitat Politècnica de València, 46022 Valencia, Spain

ARTICLE INFO

Keywords:
DSL
Microservice
Architecture
Modeling
Language
Hypergraph

ABSTRACT

Modern microservice architectures demand new features from traditional architecture description languages, many of them related to the complexity of the modeled systems. This paper first identifies common concerns found in microservice architectures. Then it presents the features required by a suitable architecture modeling language in order to face many of these concerns. Existent modeling languages get evaluated and a lightweight high-level platform-independent modeling language is proposed. The language is general enough for describing many interactive microservice architectures, bringing together most of the features found in a scattered way in previous contributions. The language is presented in an ordered way, first defining its syntax using MOF and describing informally its underlying concepts, and later proposing an alternative hypergraph-based mechanism for describing its semantics. Regarding this methodology, an architectural style gets defined using a hierarchical type hypergraph, which contains all the information about all valid software architectures in an intuitive and compact way. The feasibility of the language is then demonstrated by providing an experimental tool which translates models to different container orchestration systems. Finally, the language is evaluated against the identified features in the context of the TeaStore reference application.

1. Introduction

Software architectures have gone a long way, from classical monoliths to highly fragmented applications, based on small independent building blocks called microservices (Fowler, 2014). Microservice architectures (MSA) represent the evolution of service-oriented architectures (SOA) (Erl, 2005). They are based on the same general principles but they emerge as a refined way of decomposing systems into smaller parts, promoting different properties (Richards, 2015) (e.g. single responsibility, full autonomy, elasticity, etc.), and providing a number of benefits over previous approaches (Newman, 2015).

To implement microservice architectures, the current practice involves using container technologies (Jaramillo et al., 2016), so that each component gets executed in an independent container. A container provides a lightweight runtime environment which packages the application and all its dependencies in a fully isolated manner, perfectly matching microservices requirements.

In complex applications, thousands of interconnected microservices exist and multiple replicas of each microservice are concurrently running. Managing all these elements properly becomes an extremely complex task, which can only be achieved with success through automated means. Container orchestration systems like Kubernetes (Bernstein, 2014) help to automate most of the operations on these applications on a cluster of virtual or bare-metal machines. Consequently, the current practice involves running microservice-based applications on the most popular cloud-based container orchestrators (Pahl et al., 2019). This approach gives rise to a new generation of applications coined as cloud-native applications (CNA) (Kratzke and Quint, 2017), which focus on properties such as automation, availability, elasticity and resilience.

In this context, developers describe their microservice-based applications using the artifacts imposed by the most widespread container orchestration systems (Abdollahi Vayghan et al., 2018). These artifacts tend to be plain and help to describe an application using relatively low-level concepts, like replicated containers, connected directly or through load balancers. As a result, application definitions are rather low-level and complex, tied to concrete platforms and technologies, preventing portability and driving away inexperienced users.

Intuition dictates that having a high-level platform-independent language for describing the architecture of microservice-based applications might help to mitigate many problems. Furthermore, due to the fine-grained nature of microservice architectures, these high-level descriptions of the software actually become true maps of the application's internal structure, along with its desired state at runtime. Therefore, they prove to be very useful not only for documentation, analysis and verification purposes, but also for guiding the deployment, monitoring and management of the application at runtime, enabling model-driven engineering (MDE) approaches (Schmidt, 2006).

☆ Editor: Professor Uwe Zdun.

* Corresponding author.
E-mail addresses: jesparza@dsic.upv.es (J. Esparza-Peidro), fmunyoz@iti.upv.es (F.D. Muñoz-Escoí), josep@iti.upv.es (J.M. Bernabéu-Aubán).

https://doi.org/10.1016/j.jss.2024.112041
Received 7 September 2023; Received in revised form 27 March 2024; Accepted 28 March 2024
Available online 30 March 2024

These claims are supported by previous work in the area. To begin with, systematic mapping studies like (Soldani et al., 2018) identify the design of microservice architectures as one of the main pains of microservice practitioners, mainly due to their complexity. In this regard, works like (Di Francesco et al., 2019) point out that few papers have investigated architecture languages for microservices, identifying new research gaps, and conclude that having an industrial standard for describing the architecture of microservice-based applications might help to reason about the system and would become a powerful communication instrument between different stakeholders. Finally, recent systematic mapping studies like (Hernández-Aparicio et al., 2022; Lelovic et al., 2022) highlight a growing interest on microservice architecture languages, and demonstrate it is a hot topic in the research community. Furthermore, these studies reveal that although much work has been done in the area, most of it has been produced by the academia and there is no prominent language adopted by the industry. In this regard, previous surveys like (Malavolta et al., 2013) suggest that architecture languages incubated by the academia tend to be highly theoretical and useless for the industry, and argue that successful architecture languages should fulfill industrial needs, becoming a real communication mechanism between different stakeholders and should reach a trade-off between formalism with precise semantics and usability.

The main goal of this work is to uncover an appropriate microservices architecture modeling language which meets both academic and industrial needs. To this end, a first effort involves identifying such needs. This is the first research question presented in this paper:

- **RQ1: Which features are demanded from a modeling language for describing modern microservice architectures?**

After reviewing previous state-of-the-art surveys in the field, Section 2 identifies common concerns found in microservice architectures, both in academic and industrial contexts, and proposes in a reasoned way the features a modeling language should include for facing most of them.

A number of languages for modeling software architectures have been proposed by the scientific community in the past and all together they introduce most of the concepts required for describing microservice architectures, but in a scattered way and with noticeably different purposes. This leads to the second research question evaluated in this paper:

- **RQ2: What architecture languages may be used for modeling microservices and how do they cover the requirements identified by RQ1?**

Section 3 makes a brief walk-through of many of these efforts, and they are evaluated against the identified requirements in RQ1, yielding a comparison table. According to the present study, no prominent modeling language fully meets all the requirements. This finally leads to the third research question presented in this paper:

- **RQ3: How would look like a language meeting all the requirements identified by RQ1?**

Based on and inspired by previous contributions, this work advocates the design of a new domain-specific modeling language (DSML) (Kelly and Tolvanen, 2008) for describing microservice architectures. The benefits of using domain-specific languages have been widely studied in the scientific community (Mernik et al., 2005; van Deursen et al., 2000). In this case, the language includes high-level concepts which assist the architect to declaratively describe complex elastic configurations, where a component is composed of other components, and components get connected through different means. These components might be incarnated by lightweight containers, heavy virtual machines or any other means that is to come. Due to the abstract nature of the descriptions they could be deployed to different platforms, or a combination of them, and the platforms they are deployed onto should guarantee the properties demanded by the descriptions.

According to Harel and Rumpe (2004), describing a new language is a matter of defining its syntax and semantics in a proper way. The syntax of the language is presented in Section 4. Section 4.1 lays out the abstract syntax along with the conceptual framework it imposes. This framework borrows many concepts from previous work and puts all them together in a coherent way. It also extends some of these concepts in order to enable easy modeling and deployment of microservice-based applications. To avoid locking into a concrete syntax, a general mechanism for mapping the abstract syntax to concrete syntaxes based on serialization languages (e.g. YAML) is proposed in Section 4.2.

The semantics of the language are presented in Section 5. This section follows the approach suggested in Harel and Rumpe (2004), first describing the semantic domain of the language, and later defining a semantic mapping of the main syntactic constructs to the corresponding semantic domain objects. To formalize the semantic domain, hypergraph theory is used, and previous approaches like those described in Hirsch (2003) and Bruni et al. (2008b) are extended and combined together.

To promote and validate the proposed language two experimental tools for graphically creating models and deploying them to container-based platforms have been implemented. They are documented in Sections 6 and 7 respectively.

Finally, the language is evaluated in Section 8, in order to check whether it fulfills all the requirements identified by RQ1. To conduct such evaluation a reference microservice-based application called TeaStore (von Kistowski et al., 2018) is used.

2. Language requirements

This section presents the main features demanded from a microservices architecture modeling language. To that end, the authors first identify the main concerns and challenges faced by microservice architecture practitioners. From these common concerns emanate most of the features. Those features are presented in a reasoned way, along with a brief explanation of how they contribute to solve some of the identified challenges.

Fortunately enough, it has been a while since microservices inception (Fowler, 2014) and the problems derived from this architectural style have been widely detected in the literature. To identify them this work focuses on systematic studies conducted in the period 2016–2022, both in the academia and in the industry, which raise research questions related to the main challenges, issues, research focus or pains found in microservice-based architectures. All these works are listed in Table 1, which also determines the study focus (either academic or industrial) and the research question of interest considered within it.

Table 2 summarizes the main concerns identified in these works. For the sake of simplicity, some related concerns have been grouped into a more general category. For each presented concern, associated rationale which helps to further describe, qualify or extend it is provided. It is easy to see that each concern affects one or more phases in the microservices lifecycle (i.e. design, implementation, testing, operation). For example, concerns C1–C7 have more to do with design and implementation whereas C9 and C10 are more related to microservices operation. Even though architecture languages are mainly used in the design phase, they may include tools or aids which contribute to solve many problems related to other phases. Therefore it is reasonable to think that the language will be more effective if it addresses most of these concerns.

Once the main concerns have been presented, the next paragraphs list the main features a suitable microservice architecture modeling language should have, according to the authors. They are named from R1–R7 and they are presented along with their corresponding indicators, which further qualify them and limit their scope. Fig. 1

Table 1
Systematic studies on microservice-based architectures. Academic (A), Industrial (I).

Year	Paper - Research question	Focus
2016	A systematic mapping study in microservice architecture (Alshuqayran et al., 2016) RQ1: What are the architectural challenges that microservices systems face?	A
2017	A systematic literature review on microservices (Vural et al., 2017) RQ2: What are the main practical motivations behind microservices related research?	A
2018	The pains and gains of microservices: A Systematic grey literature review (Soldani et al., 2018) RQ2: What are the technical and operational "pains" of microservices?	I
2019	Architecting with microservices: A systematic mapping study (Di Francesco et al., 2019) RQ2: What is the focus of research on architecting with microservices?	A
2021	Deployment and communication patterns in microservice architectures: A systematic literature review (Karabey Aksakalli et al., 2021) RQ3: What are the identified issues and obstacles related to the communication and deployment of microservices?	A
2022	Challenges and solution directions of microservice architectures: A systematic literature review (Söylemez et al., 2022) RQ1: What are the identified challenges of microservice architectures?	A

Table 2
Main concerns on microservice-based architectures and how they are covered by language requirements.

Concerns	Rationale	R1	R2	R3	R4	R5	R6	R7
C1. Architecture	Large number of components. Difficulties in decomposing into units of the appropriate size.		X		X	X		X
C2. Communication	Multiple mechanisms. Multiple patterns (e.g. pub-sub, load-balancer). Compatibility between interfaces.		X		X			
C3. Storage	Data consistency, distributed transactions.		X					
C4. Service discovery	Service location, endpoint publishing.				X			
C5. Performance, scalability	Resource consumption in network and computing, response time, scheduling and predicting the load.		X	X				
C6. Fault-tolerance	Replication, availability, resiliency.			X	X			
C7. Security	Access control, large number of endpoints to protect.				X			
C8. Testing	Size of the system, execution uncertainty, benchmarking.					X		
C9. Deployment	System complexity, coordination.	X	X			X	X	
C10. Management	Operational complexity, monitoring, tracing and logging.		X		X		X	

summarizes all these requirements and corresponding indicators in a mind map.

The presented requirements contribute to solve most of the concerns previously identified. Table 2 summarizes which requirements cover which concerns. It is important to note that the identified requirements are not fully independent, they are interrelated instead.

- **R1: Platform agnostic**.

Many modeling languages are bound to specific platforms, considering a platform as any sort of system and complementary tools which host applications or services. In these cases their potential impact is limited, leading to vendor lock-in. The problem of vendor lock-in is well-known in the literature (Opara-Martins et al., 2014). One way to avoid it involves providing basic abstract concepts, useful for describing any microservice-based application. To keep the language pragmatic and easy to use, these concepts should be familiar to architects and developers, and easy to translate into common artifacts found in modern runtime platforms. This feature clearly contributes to solve deployment-related concerns (C9 in Table 2).

- **R2: Components**.

A microservice-based application is broken into many self-contained small components. Each component is intended to implement a bounded functionality, and it should include all the required resources, in a single deployable unit. Identifying these components is the first step to face any architecture-related concern (C1 in Table 2).

The functionality of every component must be published through well-defined interfaces. In addition, a component may require the functionalities provided by other components in order to meet its own goals. To properly characterize the component behavior both the provided and the required functionalities must be clearly defined. This requirement will help to identify which components can communicate and it may contribute to solve to some extent problems related to interface compatibility (concern C2 in Table 2).

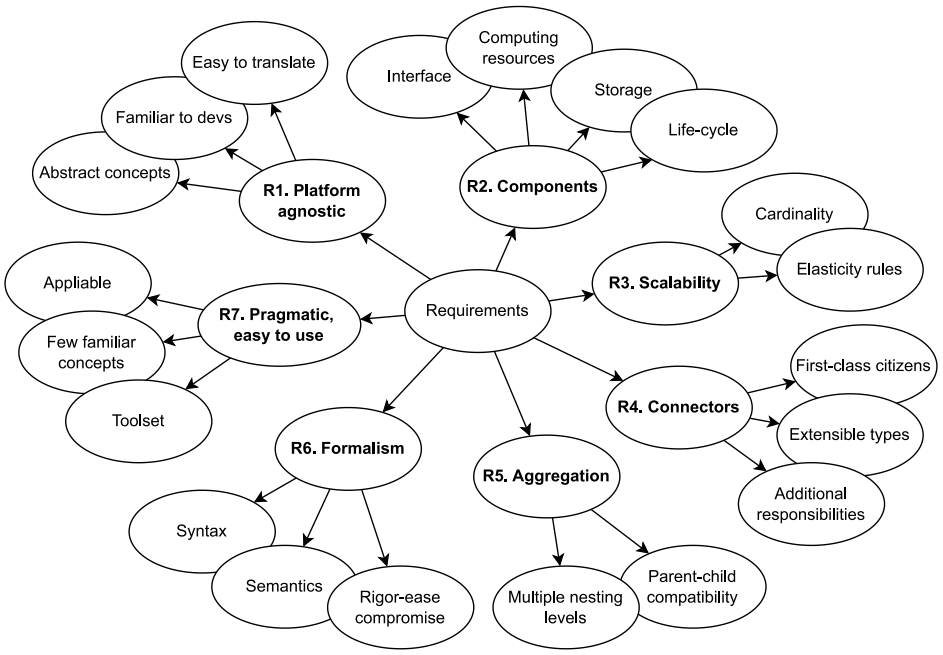

Fig. 1. Language requirements.

To provide its functionality, a microservice may require some computing resources, like CPU, memory, network bandwidth or persistent storage. Defining all these resource requirements in advance may help to schedule and predict runtime behavior, aiding with performance-related concerns (C5 in Table 2). Also, providing tools for properly modeling the persistent storage may help with some storage-related problems (C3 in Table 2).

Finally, at deployment and execution time every microservice usually goes through several phases. This process is commonly known as the microservice life-cycle, and the microservice may take different actions in each phase. Providing a way of defining and triggering this functionality may contribute to help in deployment and management related concerns (C9 and C10 in Table 2).

- **R3: Scalability**.

In modern container-based systems, many replicas of each microservice coexist. Furthermore, they get created and destroyed dynamically, considering different factors like the current load, the consumed resources, the detected failures and the system's scheduling policy.

In this scenario, in order to guarantee certain levels of quality, one should be able to define the range of available microservice replicas, as well as to configure when to increase or decrease them. This is often achieved by setting some QoS measurement-based ranges, like response time or resource consumption levels.

A language with the ability of expressing these capabilities will clearly help to solve problems related to scalability and fault-tolerance (concerns C5 and C6 in Table 2).

- **R4: Connectors**.

In a microservice-based application, microservices communicate with each other using (a)synchronous communications mechanisms. No centralized authority exists and each microservice is responsible for searching and interacting with others. In this context, it is usually said that microservices interact through *choreography* instead of *orchestration*.

In current container-based systems, the replicas of a microservice often communicate with the replicas of another microservice through a load balancer, which is a very well-known pattern, as identified in so many works (Taibi et al., 2018). It turns out that this pattern can be further generalized and extended, effectively reincarnating the old good software architecture concept of *connector* (Garlan et al., 1997) as a first-class citizen. Thereby, the replicas of a microservice may communicate with the replicas of other microservices through different connectors, and this approach has a number of benefits.

First, the connector does only enable communications between directly connected peers. This property contributes to solve access control concerns (concern C7 in Table 2) in scenarios where it is common that any microservice can connect to any other without constraints.

Second, it contributes to solve the problem of *service discovery* (concern C4 in Table 2) since the connector is responsible for keeping track of all the replicas of the target microservice.

Third, the connector routes the requests from the sender/s to the receiver/s. To that end, it may follow different strategies and criteria, implementing different connector types (or communications patterns), like the load balancer, the publisher/subscriber, the message queue or any other user-defined, effectively helping in many communications problems (concern C2 in Table 2).

Finally, the connector may take additional responsibilities related to fault tolerance and management concerns (concerns C6 and C10 in Table 2). The former may be achieved for example by implementing well-known patterns like the circuit breaker (Montesi and Weber, 2016) or fault-aware load balancers. The latter may be achieved for example by permanently monitoring, tracing or logging microservice communications.

- **R5: Aggregation**.

A microservice-based application may contain many microservices, and this is the crux of its architectural complexity (concern C1 in Table 2). It is well-known that one of the most effective tools for managing such a complex creature is by using abstraction and refinement mechanisms. Software aggregation, where a component includes other subcomponents, with multiple nesting levels, represents a common way of implementing these mechanisms and this necessarily leads to hierarchical topologies.

Introducing this feature into a modeling language is not a trivial task though, and it requires careful analysis of the interrelations between parent and children, in particular when multiple replicas of them exist. Furthermore, if microservices get connected through connectors, it is

important to determine which connections between elements are legal, and how the subcomponents of a component are reached from other components.

- **R6: Backed by a sound formalism**.

It is important to understand with precision the meaning of the language syntactical constructs. The best way to achieve this is to describe their semantics using formal theory. In addition, having a formal framework can help to build methodologies and tools for validating, managing and transforming models. Model validation tasks may help with testing activities (concern C8 in Table 2) whereas transformation tasks can help to translate high-level models to concrete platforms, tackling the deployment challenge (concern C9 in Table 2).

Nonetheless, formal frameworks are often tough and restricted to specialized communities of scientists. A compromise between rigor and ease of understanding is desirable.

- **R7: Pragmatic and easy to use**.

A language is not useful if it is not usable. To become really usable, it must be pragmatic and easy to use. To be pragmatic, it must be easy to apply to real applications. This requirement can be met by facilitating the modeling of common patterns, as well as promoting the use of best practices.

To be easy to use, the language must be simple, using a few concepts that are familiar to practitioners. In addition, these concepts should be used in intuitive accompanying toolset. One path for achieving this property may involve dissecting the tools currently used in the industry and trying to map most of their concepts to the language.

A pragmatic and easy to use language will definitely help to define complex architectures in simple terms (concern C1 in Table 2).

3. Related work

Even though much work has been done in the area of software architecture, there are still many challenges to overcome, as pointed out by some authors (Garlan, 2014). Regarding architecture modeling languages (ADL[1]), there is a large body of knowledge.

In the last thirty years it is possible to identify four main waves of ADLs, which coincidentally cover the most significant twists in software architectures to date. They are presented below, along with their indicative dates:

1. *First wave ADLs* (1990–2000): first generation of languages, which identify the main elements included in both monolithic and distributed architectures.
2. *Service Oriented Architectures* (SOA) (2000–2010): with the rise of *Service Oriented Computing* (SOC) (Papazoglou et al., 2007), the service becomes the basic piece for designing distributed applications, which follow a *Service Oriented Architecture* (Erl, 2005).
3. *Cloud computing* (2010–2016): the advent of cloud computing (Vaquero et al., 2009) demands new properties from the applications and a new way of designing software architectures.
4. *Recent efforts* (2016–today): recent efforts include languages for describing extremely dynamic architectures, like microservice-based, container-based or cyber–physical applications.

In general, it is interesting to observe a similar pattern of interest in the scientific community in every wave: after introducing a new architectural style, a growing number of modeling languages are proposed. When the field is mature enough, one ore more standardization proposals arise which collect most of the concepts and ideas raised over time. Thereafter, the number of proposals decreases dramatically.

This insight, which recalls the well-known Gartner hype cycle (Fenn and Time, 2007) and could be informally coined as the "ADL hype cycle", serves two purposes in this work:

(a) The standardization initiatives which "close" each hype cycle seem to be good representative candidates of each wave. This work will use them for evaluating the features implemented in such wave.
(b) The lack of standardization initiatives seems to be a symptom of the need to carry out more intensive work in the area. This intuition, along with some additional evidence, justify this work focused on microservice-based architectures.

To address the study of existent ADLs, when possible, previous surveys have been consulted in order to identify the most relevant work. Table 3 summarizes the obtained conclusions. It shows the four main waves along with their main representative standardization initiatives, if they exist, or the most interesting proposals for the purpose of this work otherwise. For each language, the requirements presented in Section 2 are evaluated, and the obtained results are classified into one of three compliance levels: fully supported, partially supported or not supported. The next subsections analyze in detail each wave of ADLs, and highlight the main concepts they have brought to light.

Table 3
Requirements fulfillment of different ADLs: ■ means fully supported; □ means partially supported; - means not supported.

ADL	R1	R2	R3	R4	R5	R6	R7
First wave ADLs							
UML	■	□	–	–	□	–	■
ACME	■	□	–	■	□	■	■
SOA							
SCA	■	□	–	–	□	■	■
SoaML	■	□	–	–	–	–	□
Cloud computing							
TOSCA	■	■	■	–	□	–	□
Recent efforts							
μTOSCA	■	■	■	□	□	–	□
Canonical Juju	■	■	□	–	–	–	■
Helm	–	■	■	–	–	–	■
Kumori Platform	□	■	■	■	□	–	■

3.1. First wave ADLs

Many of the languages available in the first wave are analyzed in Medvidovic and Taylor (2000). The main standardization effort in this wave is clearly embodied by the *Unified Modeling Language* (UML) standard (Object Management Group, 2017), conceived by the Object Management Group's (OMG) as a collection of diagrams for modeling different aspects of the software. In particular, the component and deployment diagram type is very helpful for breaking a system into different interconnected components (R2), with hierarchy included (R5). UML is a very generic tool for representing any kind of software entity (R1). Also, one can find many tools to create UML models (R7). However, it lacks out-of-the-box support for many features required by microservice-based architectures, like specifying the resources required by a microservice or its life-cycle events (R2), defining scalability features (R3), recognizing connectors as first class citizens (R4), constraints on replicated hierarchical topologies (R5) or presenting a clear formalism for explaining the main language constructs (R6). Over the years, some of these deficiencies have been resolved by extending UML through different profiles.

Another interesting standardization initiative in the first wave is ACME (Garlan et al., 2010), conceived as a generic architecture interchange language, which defines an ontology of concepts common

[1] ISO/IEC/IEEE 42010:2011: "any form of expression for use in architecture descriptions".

to all ADLs (R1). The existence of this language enables translation mechanisms between the artifacts used in different languages. As depicted in Table 3, it covers pretty well the requirements of the present work. Unfortunately, it was designed in a time when generalized utility computing was still far away, and most features required for the cloud, like the specification of the required resources or life-cycle events (R2), scalability features (R3), or replicated hierarchical topologies (R5) are missing. Nevertheless, ACME includes very interesting concepts like components (R2), connectors as first class citizen (R4) or support for hierarchical topologies (R5). Furthermore, the language is formalized (R6) and specialized tools exist which aid architects to create and validate specifications (R7).

In general, the languages included in this first wave did a good job in identifying the main elements of an architecture and their connections, and strove to formalize their semantics. However, they prevented expressing non-functional properties (in particular those related to cloud-related concepts like resource requirements, life-cycle or scalability) and the need to reach a compromise between formalism and its usability became clear, as pointed out in studies like (Malavolta et al., 2013).

3.2. SOA ADLs

In the case of SOA-based architectures, the Object Management Group (OMG) proposes the standard SoaML (Object Management Group, 2012), which extends UML, by including a metamodel and a profile. This specification mainly focuses on describing basic service concepts, like service consumers and providers, service contracts and choreography. It inherits some features from its parent specification (R1, R2) but it does not include specific mechanisms for describing hierarchical services (R5) and the supporting tooling is not specific (R7). So in general it is a high-level abstract specification and many of the features pursued in this work are missing as depicted in Table 3.

Another meaningful standardization effort in this wave is the OASIS Service Component Architecture (SCA) (Open SOA, 2007), which comprises a set of specifications for describing SOA-based applications. In particular, the SCA Assembly Model is an abstract model (R1) which describes how distributed components (R2) connect with each other in a composite (R5) application. There is specific tooling available (R7) and some formalization efforts (R6) can be found in the literature. This specification is very aligned with this work vision, but again it does not consider cloud related concepts that have to do with the required resources, life-cycle (R2) or scalability features (R3). Furthermore, even though hierarchical topologies (R5) are supported, due to the lack of scalability features, it is not clear how replicated components connect with each other in such architectures. Finally, connections between components are represented with "wires", so connectors are not considered as first class citizens (R4).

In general, the main focus of the languages included in this wave is to define service contract and composition. In addition, in many cases contract specifications are enriched with QoS measurements (Tran and Tsuji, 2009), related to attributes such as performance, security, reliability, etc. These measurements represent the non-functional guarantees provided by a service, also known as Service Level Agreements (SLAs). Finally, it is curious to note that despite the existence of standards, most practitioners still use UML as the main language for modeling SOA architectures, as pointed out in Zaafouri et al. (2021).

3.3. Cloud ADLs

The emergence of cloud computing imposes new rules, constraints and properties to applications, which make more classical approaches for modeling applications not appropriate. This situation motivates more research on languages for modeling cloud-based applications, like *BluePrint*, *CAML*, *CloudML*, *MOCCA*, *MULTICLAPP*, etc. Most of these languages are analyzed in Bergmayr et al. (2018). They focus on modeling the topology of the application, the services and resources offered by the cloud platform and some non-functional properties related to the elasticity or the service level (e.g. latency, reliability, security, etc.) required by the application components. In general, they spend too much effort in defining the structure, allocated resources and deployment targets of heavy-weight virtual machines.

In this wave, the most important standard agreed upon is OASIS TOSCA (OASIS, 2013b). This proposal focuses on describing the topology of a cloud application along with the processes required for automating its life-cycle operations in a portable manner. Starting with very abstract (R1) concepts – services, nodes and relations – one can describe almost anything, even complex topologies (R2) in a hierarchical way (R5). Also, there are tools (R7) available for specifying and deploying TOSCA-based diagrams. According to some authors (Lipton et al., 2018), TOSCA with additional customizations is the way to follow for modeling microservice architectures in the future. However, it is not a popular choice among microservice architects, as pointed out in Di Francesco et al. (2019). Some of the reasons might be found next:

- Even though TOSCA was conceived for describing cloud applications (R3), it is mainly used for describing infrastructure, as it can be seen in most of the available examples (OASIS, 2013a).
- No formal definition of its semantics is provided (R6), giving rise to different interpretations. For example, regarding hierarchical topologies, an informal "matching" concept between service templates and node types is provided. This has motivated that some authors (Brogi and Soldani, 2013) elaborated their own notion of formal "matching".
- TOSCA considers the concept of relationship for modeling connections between nodes. These relationships are typically used for defining dependencies (through capability–requirement pairs) or deployments (where a component gets installed into a particular environment). Thus, the concept of a connector as a first class citizen (R4) is blurred and must be simulated through a combination of relationships and nodes, making more difficult both its specification and understanding.
- TOSCA specifications are too generic. A node may represent a machine, a hosting environment, or an application. A relationship may represent a dependency (e.g. a library dependency, a remote service dependency), or a component–container relationship (i.e. a component gets executed in a particular container execution environment), or any other association between two nodes. Therefore, describing and interpreting a model is not a simple task (R7), since the meaning of every element requires additional explanations. In this sense, TOSCA feels more like a general purpose language, capable of specifying any element and any type of relation between elements. This approach hinders the main advantages of DSLs which involve achieving higher accuracy in the specifications by using more expressive constructs which save time and effort, as many studies demonstrate (Wile, 2004; Kosar, 2010; Barišić et al., 2011).

3.4. Recent efforts

In recent years, the need to properly model microservice-based applications has become apparent. This situation has led to new activity in the area coming from two main sources: the academia, proposing microservice ADLs, and the industry, proposing platform-specific tools and languages.

Regarding the academia, studies like (Hernández-Aparicio et al., 2022; Lelovic et al., 2022) collect most of the available initiatives. According to these studies, no prominent language exists, and concrete languages are proposed in the context of solving concrete problems, but no holistic approach is usually taken. Most of these languages focus on describing microservice behavior and generating automatically part of the application, so they do not pursue the goals considered by

the present work. Interestingly enough, TOSCA remains as a viable alternative.

In this regard, extensions like μTOSCA (Brogi et al., 2020) attempt to solve some of the difficulties posed by this standard for modeling microservice-based applications. Being more specific, μTOSCA provides a type system useful for modeling services, communication patterns and databases. Therefore, it inherits all features from its parent specification (R1,R2,R3,R5,R7) and identifies the need to model connectors as first class citizens (R4) but it does not elaborate on hierarchical topologies (R5) other than the tools provided by the basic language. Furthermore, to become proficient in the language one must first master TOSCA, to be able to use this new type system, hindering ease of use (R7).

Regarding the industry, some solutions with the aim of simplifying the deployment of complex cloud-based applications have become very popular. Two representative initiatives worth mentioning are Canonical Juju (Canonical Ltd, 2012) and Helm (The Linux Foundation, 2017). Even though they do not provide an architecture description language in the classical sense, they include tools (R7) for describing components (R2) and dependencies in an easy way, demonstrating how simple efforts in this direction bring great benefits to users.

Canonical Juju (Canonical Ltd, 2012) aims to provide a model for describing software topologies in an abstract way (R1). To that end, Juju makes use of an artifact called charm for modeling scalable services (R3). Multiple charms are available in a public store and they can be connected together for creating bigger architectures, which can be described within bundles. Even though advanced features like elasticity (R3), custom connectors (R4) or hierarchy (R5) are not fully supported, and formal semantics are not specified (R6), it is a great example about how a simple architecture description can help users to deploy complex applications in an easy way.

Helm (The Linux Foundation, 2017) self-describes as a package manager for Kubernetes. The packages are called charts, and they contain all the information required to create a Kubernetes application, in essence a collection of files that describe a related set of Kubernetes resources. A chart is configurable, and may depend on other charts, defining a topology of dependencies. Again, connectors (R4) and hierarchical structures (R5) cannot be managed explicitly, but elastic features (R3) like auto-scaling may be defined using Kubernetes artifacts. Finally, it is obvious that this tool is strongly linked to the Kubernetes platform, so it is not a platform-agnostic solution (R1).

Another relevant industrial initiative can be found on the Kumori Platform (Kumori Systems SL, 2020). It publishes a model for specifying microservice-based applications. This model identifies microservices, as a collection of running instances (R3) of components (R2), and services, as collections of microservices connected through connectors (R4) with different semantics (load balancer vs full connector). Even though the language is generally platform-agnostic, for now it just focuses on Docker images and Kubernetes (R1). Multiple hierarchical levels are not supported (R5) and there is no accompanying formalism for describing the model semantics (R6).

4. The language syntax

As stated in Chen et al. (2005) a DSML can be formally defined as a 5-tuple, $L = \{C, A, S, M_S, M_C\}$. C is the concrete syntax, which specifies the notation used to express models. A is the abstract syntax, which defines the concepts, relations, attributes and integrity constraints available in the language. S is the semantic domain, which typically includes some mathematical objects whose meaning is well-defined. $M_S : A \rightarrow S$ is the semantic mapping, which assigns syntactic concepts to objects included in the semantic domain S. Finally, $M_C : C \rightarrow A$ is the syntactic mapping, which assigns syntactic constructs to objects in the abstract syntax A.

According to Medvidovic and Taylor (2000), any architecture description language imposes a conceptual framework which includes a set of concepts, relationships and rules. Section 4.1 presents the abstract syntax of the language A along with a brief informal description of all the elements proposed by the language (i.e., its conceptual framework), also including intuitive box-and-line graphical notation.

All the elements identified in the abstract syntax can be easily described using simple data types (strings, numbers, etc.) and data structures (dictionaries, arrays, etc.). Section 4.2 discusses how these elements may translate to popular serialization languages.

4.1. The abstract syntax

A common technique for specifying the abstract syntax of a DSML is to use metamodeling (Bryant et al., 2011). To that end, another modeling language is used to describe the concepts, relations and constraints of the DSML. MOF (Meta Object Facility) (Object Management Group, 2016) is a standard promoted by OMG for specifying the syntax of other languages, based on a simplified version of the UML2 (Object Management Group, 2017) class diagram. A simplified version of the modeling language abstract syntax is shown in Fig. 2 using MOF. The next paragraphs introduce the main elements found in the diagram.

- **Basic components**.

An application is made of *components*. Components connect to each other creating complex configurations. A component may be considered a black box which provides some functionality to others. Fig. 3 shows three basic components as simple boxes: a web server www, a database server db and a log server log.

Components are scalable, and present a given *cardinality*, which means many replicas of each component may run concurrently at execution time; they are called *component instances* (or just *instances*). Every instance runs the bits of a package obtained from a given *source* (i.e. a URL) in a particular *runtime* (e.g. Docker, VBox, Java, etc.). Every instance has its own identity (unique identifier, unique address, etc.) and requires some *resources* to run (e.g. CPU cores, memory, etc.). Furthermore, instances get started/stopped dynamically in order to ensure an appropriate overall component performance. Hence they present an elastic (Herbst et al., 2013) behavior, which may be configured through appropriate *auto-scaling policies* (Lorido-Botran et al., 2014). In Fig. 3 components www and log are fully elastic, they do not present *cardinality* bounds (cardinality is specified using the syntax $[min : max]$). The component db represents a scalable database with a number of replicas included in the range $[1 : 5]$.

A component may be further configured using *variables*. A variable is a pair (name, value) which is available within instances at runtime. When an instance gets started, its variables are typically consulted in order to configure several aspects of its operation.

A component may publish an interface for enabling communications with other components. To that end, it defines a collection of *endpoints*. Every endpoint may be either *in* or *out* and supports a particular *protocol*. An in endpoint models inbound connections and typically publishes some functionality that will be consumed by other component/s. An out endpoint models outbound connections and typically represents a dependency of the component. An out endpoint must connect to a protocol-compatible in endpoint of the same or another component. In Fig. 3, component www has one in and one out endpoints, component db has two in and two out endpoints and component log has one in endpoint.

Regarding component instances *durability*, a component may be either *ephemeral* or *permanent*. Ephemeral instances are stateless; when they stop or fail they are automatically discarded, along with all their allocated resources. Conversely, permanent instances are stateful, when they stop or fail, their state (identity, assigned address, variables, volumes) gets preserved until the instance is restarted again. In Fig. 3 components www and log are ephemeral, and component db is permanent, since it is modeling a master–slave configuration of a replicated database.

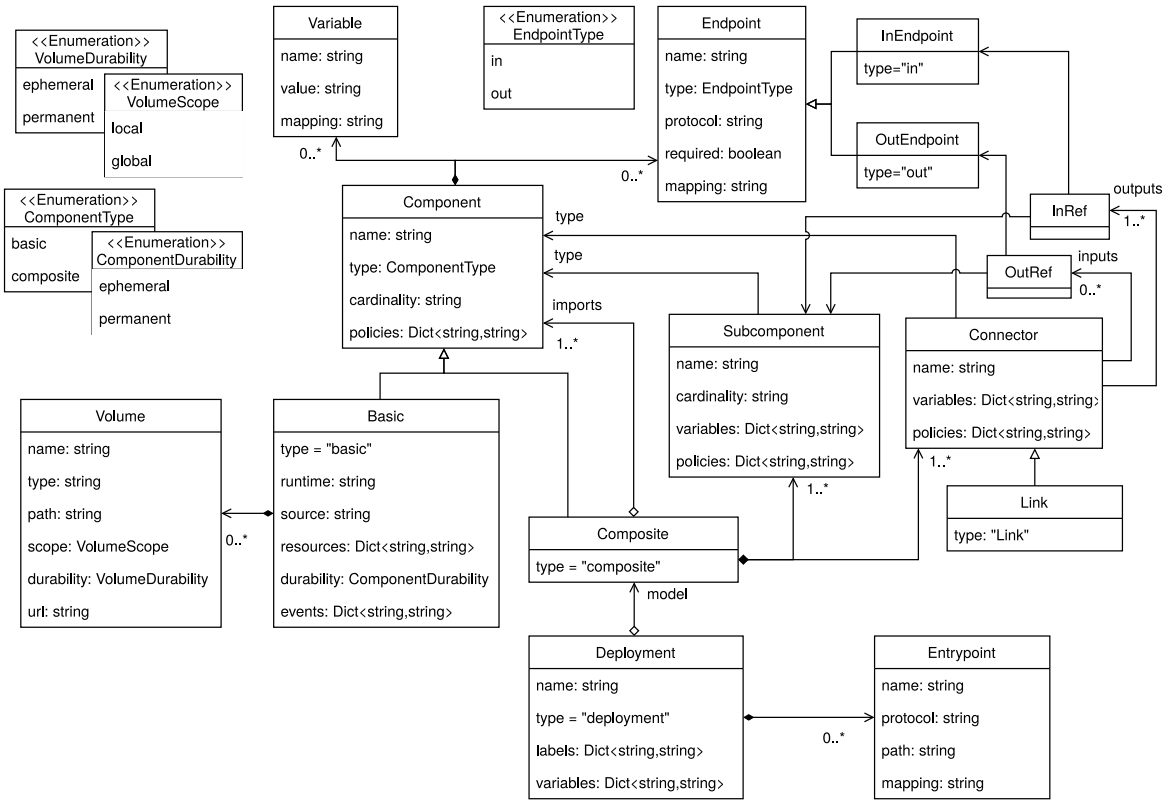

Fig. 2. Abstract syntax described using MOF.

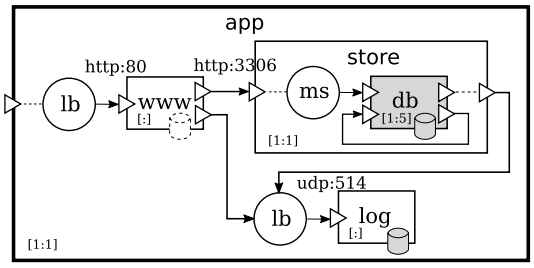

Fig. 3. Graphical model sample.

Finally, instances go over several states throughout their life cycle, and due to the dynamic nature of the system, their context (the instances they are connected to) changes very frequently. These *events* must be communicated to instances, so that they can take appropriate measures. To that end, components can register handlers or listeners, which execute certain scripts when events are triggered.

- **Volumes.**

Volumes of different *types* may be attached to components. At runtime, they get incarnated by (physical or virtual) disks, which are mounted on a given *path*, and component instances access them as regular file systems.

Every volume has a particular *scope*, which may be either *local* or *global*. If the volume is local, each instance receives its own private storage area. If the component is scalable, at runtime an array of disks gets actually created, and each disk is assigned to a different instance. If the volume is global, then the same disk is shared by all instances, and remains accessible through a given *url*.

If the volume is global-scoped, its life cycle is managed externally. Otherwise, if the volume is local-scoped, then its *durability* may be either *ephemeral* or *permanent*. The life span of a local-scoped ephemeral disk is bound to the attached instance life cycle. Conversely, the life span of a local-scoped permanent disk is independent from the life cycle of the attached instance. In this case, a reusable array of disks gets created by demand.

Fig. 3 shows three volumes (as cylinders): a global-scoped volume (represented with dashed line) attached to component www and two local-scoped (represented with continuous line) permanent (represented with gray background) volumes attached to components db and log.

- **Connectors.**

The endpoints of different components get connected through *connectors*. At runtime, connectors address communications between the instances of the related components, following different delivery strategies. A connector connects an out endpoint of a sender component to an in endpoint of a receiver component. In order to successfully connect two endpoints together both their types and protocols must be compatible.

Two main connector types are supported:

- *Regular connector*: this connector is made accessible to the sender instances through a single (virtual) address, effectively hiding the instances of the receiver counterparts. It may put in practice different delivery strategies. It may be provided natively or implemented through a regular component. Some examples of common regular connectors are: the load balancer connector, the master–slave connector, the pub-sub connector, etc. In Fig. 3, regular connectors are represented with circles. The components www and log are made accessible through a load balancer, whereas the component db is made accessible through a master–slave connector.
- *Full connector* or *link*: it connects the instances of adjacent components together, with no restrictions. The sender instances have

full visibility on the receiver instances, and they may be addressed individually. In Fig. 3 a link (modeled with a simple arrow) connects components *www* and *store*.

Each domain may support different types of connectors through genuine native constructs. However, all domains must support at least the full connector and enable an extension mechanism for defining new connector types out of regular components. This mechanism must include the ability to hide all the component instances which implement the connector at runtime behind a virtual address.

- **Composites**.

A *composite* is a compound component which includes several interconnected subcomponents (either basic or composite) and is managed externally as a basic unit. A composite is a very powerful architecting tool which allows to hide the complexities of a subsystem behind a reusable, replicable unit. In this way, a complex system gets decomposed into multiple independent subsystems, enabling the specification of architectures with multiple levels of refinement. Fig. 3 is actually the representation of the composite *app*. This composite includes two basic subcomponents, two connectors and one composite subcomponent *store*, which gets further refined into one basic subcomponent and one connector.

A composite is also a *scalable* component and multiple instances may execute at runtime. In Fig. 3 a single instance of both composites exists. A composite instance is a virtual entity which enables a new independent execution context for all internal subcomponents and connectors. When a composite instance gets created, instances of all internal subcomponents get created too. When a composite instance gets destroyed, all the resources allocated by its internal subcomponents get destroyed too. Therefore, a composite instance does not directly allocate resources, but indirectly through its internal subcomponents and connectors.

Variables may also be defined on composites. In such case, they must be mapped to internal subcomponents variables. Composite variables initial values may be inherited from the mapped internal subcomponents or otherwise overwritten by the composite specification.

A composite may also publish an interface by defining a collection of *endpoints*. An *in* endpoint models a functionality exported by an internal subcomponent, whereas an *out* endpoint models an unresolved dependency of an internal subcomponent. At runtime, every composite in endpoint must be externally observed as a single address, and every incoming connection must be redirected to the *in* endpoint of a subcomponent instance. To enable this behavior, all incoming connections must be forwarded to an intermediate regular connector, which distributes the load among the connected subcomponent instances. In Fig. 3 the composite *app* in endpoint is served by a load balancer connector, which forwards the load to the *www* subcomponent instances. Likewise, the composite *store* *in* endpoint is served by a master–slave connector, which forwards connections to the master instance of the *db* subcomponent. Finally, the *store* composite publishes an out endpoint which forwards all outgoing connections coming from *db* subcomponent instances to the *log* component.

- **Deployment**.

A *deployment* represents the creation of a composite instance (typically modeling an independent application) on one or multiple target platforms. Before creating the instance, it may be further configured, setting the *variables* exposed by the composite. In addition, the in endpoints published by the composite can be made externally available, the so-called *entry points*.

4.2. The concrete syntax

In this section both the concrete syntax C and the syntactic mapping $M_C : C \rightarrow A$, which assigns every structure defined in the concrete syntax C to objects in the abstract syntax A are presented.

Rather than defining a new singular syntax for the modeling language, and specifying it using a formal notation like EBNF or similar, the strategy followed in this work involves first determining the main data structures required for describing all the entities, properties and relations identified in the abstract syntax, and later mapping them to current data serialization languages in a coherent and predictable way. Thereby the concrete syntax of the modeling language might easily evolve throughout time, adapting to new standards and supporting technologies, taking advantage of best of breed parsers and validators.

In general, the data structures required for describing a model are obtained by applying the following simple rules to the abstract syntax: classes are represented using dictionaries or structs. Single-valued properties are represented using basic data types like strings (including enumerations) and booleans. Multi-valued properties are represented with string dictionaries/structs. Associations are represented using a single string for single-valued references and using a string dictionary/list/array for multi-valued references; in any case, the string value contains the name of the referenced entity. Compositions (black diamond) embed the associated single-valued or multi-valued entities using a single dictionary/struct or a dictionary/struct of dictionaries/structs respectively. Aggregations (white diamond) are more flexible and can behave like compositions or associations, using embedding or references as needed. Notice the selected data structures are rather conventional and may be easily implemented using any modern programming or serialization language. The language just needs minimal support for the notions of string, boolean, dictionary/struct/object and list/array.

In addition, to make the concrete syntax highly modular, the specification of a complete model can be broken into smaller meaningful definitions which may either coexist independently of each other or be combined into a single definition. For enabling further reusability, these definitions may be linked or embedded as needed. With this idea in mind one can identify three main entities in the abstract syntax eligible for being independently defined: basic components, composites and deployments.

For the sake of example, and following the previous rules, a concrete syntax using JSON/YAML (ECMA Int, 2017; YAML Language Development Team, 2021) is proposed in this work. The specification of such a concrete syntax is given in the form of JSON Schema (JSON Schema, 2020) and is publicly available.[2] With these specifications other researchers may easily consult the syntax and use the schema for validating their own models.

5. The language semantics

In this section the basic semantics of the language are described. To that end, denotational semantics (Slonneger and Kurtz, 1995) are used, mapping the main constructs of the modeling language to well-known mathematical objects. In the next sections, first the semantic domain which includes all the required mathematical objects is presented and then the semantic mapping is described.

[2] https://github.com/jaespei/komponents/tree/main/schemas.

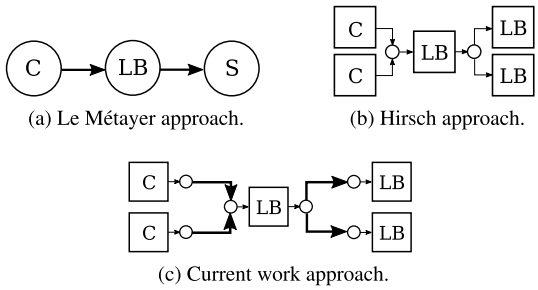

Fig. 4. Different representations of graph-based software architectures.

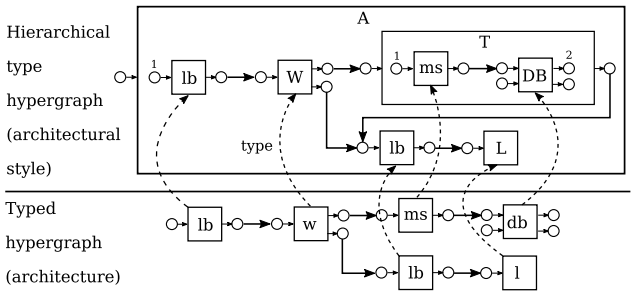

Fig. 5. Hierarchical type hypergraph and sample typed hypergraph.

5.1. The semantic domain

One can find multiple methods for formalizing software architectures (Allen, 1997; Bradbury et al., 2004): process algebras, logic-based formalisms, graphs, etc. Graph grammars are specially appealing for modeling and reasoning about software architectures: on the one hand graphs are intuitive enough since they evoke a clear picture of the system structure; on the other hand they represent a well-established formal framework based on consolidated theory. They were originally defined in Ehrig et al. (1973) and have been extended by many contributions and applied to multiple fields. When modeling software architectures with graph grammars, one of two general approaches is typically followed in the literature: the Le Metayer (1998) approach or the Hirsch et al. (1998) approach.

In the first approach, the general well-known concept of *graph* $G = (V, E)$ is used, where components are represented by the nodes V and communications links are represented by the arcs E. An example with a client, a load balancer and a server is presented in Fig. 4(a). It is simple and intuitive, and allows to model system topologies in a natural way, easily documenting the main elements of the architecture and their interconnections.

The second approach makes use of *hypergraphs* (V, E, att) (Habel, 1992; Drewes et al., 1997), where the *hyperedges* (E) can connect more than two vertices (V). A hyperedge is an atomic item with an ordered set of tentacles, which get attached to nodes (attachment nodes) through the mapping $att : E \to V^*$. Analogously to regular edges, hyperedges may be either undirected or directed. With this approach, components are represented by hyperedges and the communications ports are represented by nodes. If two hyperedges share a node, that means there is a communications link between both components. Therefore nodes act as some sort of communications hub. An example is depicted in Fig. 4(b).

The authors of this work consider the second approach more expressive, since both the topology and the external interface of the components can be defined within the same formal framework. However, hereafter a slightly different technique is taken (see Fig. 4(c)): each component is modeled as a directed hyperedge with as many source/destination attachment nodes as in/out endpoints. Then components get connected using additional directed arcs, which join together an out endpoint of a component with an in endpoint of another component. With this approach an attachment node models an in/out endpoint of the component and it is part of the component definition. Also, connectors between components are explicitly modeled with arcs. Note in Fig. 4(c) how thin and small arrows represent the hyperedge tentacles and thick long arrows represent regular edges with one source node and one destination node.

Architectures with similar characteristics can be grouped into classes or families, called *architectural styles* (Garlan and Shaw, 1993) (e.g. client–server architectures, pipeline architectures, etc.). An architectural style includes all the architectures exhibiting a common pattern and it determines which components can be included in the architecture and how they can be legally interconnected. One can think of an architectural style as a type and an architecture as an instance of that type.

One way of defining architectural styles is documented in Bruni et al. (2008a). In this approach an architectural style is defined using a *type hypergraph* $T = (V, E, att)$ (Ehrig et al., 2006), which describes the types of components and communications ports permitted. Then an architecture complying with such style is a typed hypergraph $(H, type)$, where the homomorphism $type : H \to T$ maps components in H to component types in T. This approach is very intuitive, and nicely matches the idea of an architectural style/architecture as a type/instance, but does not consider hierarchical architectures.

Combining all these previous contributions with hierarchical hypergraphs (Drewes et al., 2002), it is possible to describe an architectural style using a **hierarchical type hypergraph** $(V, E, att, port, lab, F, cts)$, where (V, E, att) is a type hypergraph, $port : \mathbb{N}_+ \to V^*$ is a mapping (Courcelle et al., 1993) identifying the endpoints published by a component, $lab : E \to T \cup N$ is a mapping labeling each hyperedge to a (non)terminal symbol. Terminal symbols $a \in T$ represent connectors whereas nonterminal symbols $A \in N$ represent scalable components. $F \subseteq E$ contains special edges called *frames*, which are mapped by $cts : F \to \mathcal{H}$ to hierarchical type hypergraphs, where $\mathcal{H}$ is the set of all hierarchical type hypergraphs. In this representation, $f \in F$ models the abstract view of a composite, whereas $cts(f)$ models the refinement of such composite. To reconcile the abstract view of the composite with its refinement, the number of attachment nodes of f must match the number of ports in $cts(f)$. The mapping is assumed to happen in clockwise direction.

An example of this approach is illustrated in Fig. 5. At the top of the figure one can find a faithful representation of the model presented in Fig. 3, now as a hierarchical type hypergraph, representing an architectural style. The hyperedges A and T are frames (composites). $cts(A)$ and $cts(T)$ map to the hypergraphs included within A and T respectively. Note A has a single attachment node, which matches the single port defined in the hypergraph $cts(A)$, labeled with 1. Likewise, T has two attachment nodes which match the ports defined in $cts(T)$ with labels 1 and 2. In addition, scalable components are represented by nonterminal symbols (uppercase letters), whereas connectors are represented by terminal symbols (lowercase letters). This hierarchical type hypergraph contains the essential information about all permitted architectures in a very compact way. For example, at the bottom of the figure one can find a typed hypergraph $(H, type)$, where the homomorphism $type$ is represented with dashed arrows (for simplicity, mappings of regular edges are omitted). This typed hypergraph represents a concrete software architecture complying with the given architectural style.

5.2. The semantic mapping

Once the semantic domain has been fixed, the main constructs of the abstract syntax presented in Section 4.1 must be mapped to the elements included in the semantic domain.

As discussed in previous sections, the highest level construct identified in the abstract syntax is *Deployment*. But *Deployment* just points to a

Composite specification. A *Composite* specification defines all the valid configurations of a collection of component instances and connectors at runtime. Therefore, it specifies an architectural style in the terms described in Section 5.1 and thus it can be characterized through a single hierarchical type hypergraph. In the next lines a procedure for building a hierarchical type hypergraph out of a composite specification is presented.

Let $X = (S, C)$ be a composite specification, where S represents the collection of subcomponents and C represents the collection of connectors. Let us assume every subcomponent $s \in S$ (connector $c \in C$) is an instance of *Subcomponent* (*Connector*) (see Fig. 2), and inherits all its attributes. Then a hierarchical type hypergraph $AS = (V, E, att, port, lab, F, cts)$ can be built following the next rules:

1. For each subcomponent $s \in S \mid s.type.type =$ "*basic*" a new hyperedge $e \mid lab(e) \in N$ is added to E. For each $ep \in s.type.endpoints$, a new node v is added to V. If $ep.type =$ "*in*" ($ep.type =$ "*out*") then v will be an input (output) attachment node of e.
2. For each link connector $c \in C \mid c.type =$ "*Link*" a new hyperedge $e \mid lab(e) \in T$ is added to E. The hyperedge e has a single input and output attachment nodes, v and w, which are the result of mapping the endpoint references $c.inputs[0]$ and $c.outputs[0]$ to AS respectively.
3. For each regular connector $c \in C \mid c.type \neq$ "*Link*", a new hyperedge $e \mid lab(e) \in T$ is added to E. In addition, two new nodes v and w are added to V and they are set as input and output attachment nodes of e respectively. Then for each endpoint reference $in \in c.inputs$ a new hyperedge $e' \mid lab(e') \in T$ is added to E. The hyperedge e' has a single output attachment node v and a single input attachment node v', which is the result of mapping the endpoint reference in to AS. Finally, for each endpoint reference $out \in c.outputs$ a new hyperedge $e'' \mid lab(e'') \in T$ is added to E. The hyperedge e'' has a single input attachment node w and a single output attachment node w', which is the result of mapping the endpoint reference out to AS.
4. For each subcomponent $s \in S \mid s.type.type =$ "*composite*", a new frame f is added to F. $cts(f)$ maps to a hierarchical type hypergraph $H = (V', E', att', port', lab', F', cts')$ following all these rules recursively. In addition, for each $ep_i \in s.type.endpoints$, a new node v is added to V. If $ep_i.type =$ "*in*" then v will be an input attachment node of f, and $port'(i) = \{w\}$, where $w \in V'$ is the input attachment node of $e' \in E'$, which is the result of mapping the connector reference $ep_i.mapping$ to AS. Finally, if $ep_i.type =$ "*out*" then v will be an output attachment node of f, and $port'(i) = \{w\}$, where w is the result of mapping the output endpoint reference $ep_i.mapping$ to AS (an output attachment node of a component $e' \in E'$).

The first rule describes how every basic subcomponent gets mapped to a regular hyperedge in AS. The second rule describes how every link connector gets mapped to a "connection" hyperedge to AS, binding together the source and destination subcomponents. The third rule describes how every connector different than link gets mapped to a regular hyperedge, and as many "connection" hyperedges as required for binding together the different peers. Finally, the fourth rule describes how every composite subcomponent gets mapped to a frame, and how the frame contents connect with the frame endpoints.

To illustrate this procedure, it is easy to see how the model presented in Fig. 3 gets mapped to the hierarchical type hypergraph displayed at the top of Fig. 5.

6. The modeling tool

As stated in Medvidovic and Taylor (2000), ADLs should not only provide formal syntax and semantics. To become really useful they should also provide some level of support to developers, typically through graphical tools which ease the edition of models. To that end, a graphical modeling tool for assisting the developer to define diagrams has already been created. A screenshot of this tool is presented in Fig. 6, which shows the sample model introduced in Fig. 3.

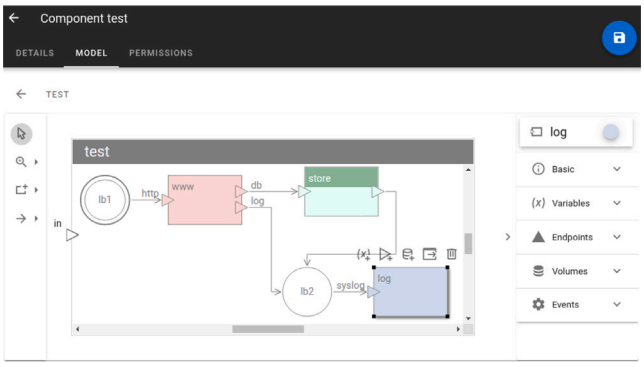

Fig. 6. Screenshot of the modeling tool.

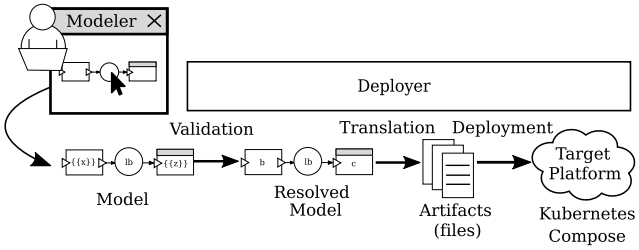

Fig. 7. Model-driven methodology covered by the experimental tools implemented in this work.

It is a web-based tool where multiple users may define and share their models. In editing mode, they are able to select components and connectors from a palette and drop them onto a canvas. Composites can be refined by double-clicking on them. Previously defined components can be easily reused in more complex architectures. The tool checks the models are well-formed and guides the user to conform with the language rules. Once the model is finished the user may export it, obtaining a specification in YAML.

7. Validation

To demonstrate the feasibility and usefulness of the proposed language an experimental Deployer tool[3] has been developed, and it has been used on several sample models. This tool takes as input a model in JSON/YAML format and is able to deploy it to different target platforms. To that end, the source model goes through a well-defined transformation procedure including three main steps, namely: validation, translation and deployment. By using the modeling and deployment tool, the full life-cycle of a model-driven methodology gets covered, as depicted in Fig. 7. The following sections describe with more detail the phases implemented by the Deployer tool.

7.1. Model validation

The first step in the transformation process involves checking that the model fulfills the syntax presented in Section 4. At the same time, the model is resolved. This means (i) propagating all the variables defined in high level components (composites) down the hierarchy

[3] https://github.com/jaespei/komponents.

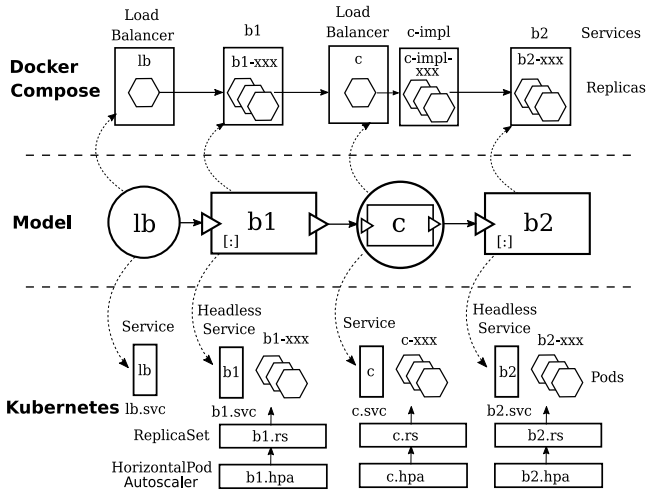

Fig. 8. Example of model translated to both Kubernetes and Docker Compose.

and (ii) evaluating dynamic expressions (they are represented with the syntax {{ expression }} and may appear in many model attributes). The output of this phase is a resolved model.

7.2. Model translation

The resolved model is used for generating artifacts supported by different target platforms. These artifacts must implement the expected model semantics. As of today, it is possible to translate models to Kubernetes[4] and Docker Compose.[5] Both of them are popular container orchestration frameworks, but they take different approaches which meet different needs. The following lines briefly describe their main characteristics.

Kubernetes is a platform for running containerized workloads on a cluster of servers. To that end it exposes different resource types the user can manage to leverage its services and functionalities. A Pod represents the smallest deployable unit, it includes a set of highly coupled containers, and it is often used for implementing microservices. A ReplicaSet is a controller responsible for keeping a given number of replicated Pods continuously running, automatically starting or stopping them as needed. An HorizontalPodAutoscaler is a controller which monitorizes a collection of Pods and decides the number of replicas which should be running in order to fulfill some resource consumption requirements. Finally, a Service keeps track of a collection of Pods and hides them under a given DNS name which resolves to a fixed IP address. Then the incoming connections are automatically redirected among the hidden Pods using load balancing capabilities.

Docker Compose is a tool for defining and running multi-container applications on a single server. All the application components are defined in a specification, and Compose automatically manages them. The main artifact managed by Compose is the service, which stands for a collection of containers created with the same configuration. Even though Compose allows to scale services creating multiple replicas it lacks more advanced features like automatic scaling or load balancing. It is primarily used in development and testing environments, and not in production, where tools like Kubernetes or Docker Swarm are more suitable.

The artifacts provided by these two systems have proven to be really useful in the translation process. An illustrative example of such a procedure is presented in Fig. 8. In the middle of the figure there is a sample model including two basic components *b1* and *b2*, and two connectors *lb* and *c*. The *lb* connector is native, and thus an implementation must be provided by the underlying platform. The *c* connector is user-defined and implemented by the basic component *c*. The bottom of the figure describes how each element gets translated into Kubernetes artifacts whereas the top of the figure does the same thing with Docker Compose. To better understand the procedure, the key guidelines applied in the translation process are summarized in the following paragraphs. For the sake of brevity, the presented discussion focuses on translating the main concepts (components and connectors) and omits others (volumes).

- **Basic components.**

Basic components represent the cornerstone of microservice-based applications since they ultimately contain the logic of the microservices. This logic should be agnostic with regard to the target platform. To that end some conventions must be followed in order to guarantee that the same piece of code will seamlessly run in every platform. The main assumptions about basic components taken in this work are listed next:

- Basic component instances are implemented as containers. Only code packaged within Docker images is supported.
- At runtime, model variables will be available in the container as environment variables.
- At runtime, model out endpoints must ultimately resolve to the actual IP addresses of the receiver peers. To that end, for each out endpoint with name *XX*, three environment variables *XX_DNS*, *XX_PORT* and *XX_PROTOCOL* containing the details of the endpoint are made available. The variable *XX_DNS* will contain a DNS name which resolves to the IP addresses of the receiver peers using regular DNS resolution mechanisms.

In Kubernetes, the instances of a basic component are implemented as Pods and each Pod receives the configuration stated in the model. The mapping of most attributes is almost trivial. For example, the component source becomes the image of a single container running within the Pod, the variables are mapped to environment variables, etc.

To create and keep all the instances of a basic component running a ReplicaSet gets generated. If the cardinality of the component includes a range, then the ReplicaSet is initialized with the lower limit and a HorizontalPodAutoscaler is generated too. The latter keeps the number of running replicas within the given limits, taking into account the specified resource consuming policies.

Finally, for discovery purposes an up-front headless Service is also generated. In Kubernetes headless Services do not provide load balancing capabilities but they allow to keep track of a collection of Pods. To that end, for each Pod backed by the Service an A record with the name of the Service and the IP address of the Pod gets registered in the DNS server. For this reason, the name of the headless Service resolves to multiple IP addresses, one for each backed Pod.

Fig. 8 shows how the basic components *b1* and *b2* give birth to these three resources, each one defined within its own specification file, which receives the name of the component as prefix and the resource type as suffix (.svc for Service, .rs for ReplicaSet and .hpa for HorizontalPodAutoscaler). The created Pods keep as prefix the name of the component and as suffix a randomly generated identifier. Also note how the headless Service takes the name of the basic component and thus it can be resolved to the addresses of the backed Pods.

Regarding Docker Compose, a basic component is implemented with a service (not to be confused with Kubernetes Services). As stated before, Compose does not provide automatic scalability tools so a fixed number of service replicas must be created. To aid instance discovery, services define an alias for their replicas which match the component name. The effect of defining aliases is very similar to the headless Service in Kubernetes, the DNS name of the component resolves to

[4] https://kubernetes.io/.
[5] https://docs.docker.com/compose/.

multiple IP addresses, one for each service replica. Fig. 8 shows how the two basic components *b1* and *b2* get translated to two services with the same names. When Compose creates service replicas they receive as name the prefix of the service and as suffix a sequential number automatically generated.

- **Link connectors**.

Links allow to directly connect the instances of two components. This means the instances of the sender component know about the IP addresses of the instances of the receiver component, and can freely connect to them.

In a Kubernetes cluster, all Pods can communicate with each other so no additional connectivity mechanisms must be deployed. However, as mentioned above, for discovery purposes the DNS name of the receiver component must be made available to the sender Pods in the form of environment variables. This should be easy in simple cases, when the receiver component is basic, since an up-front headless Service should exist, and the DNS name of the basic component should resolve to the IP addresses of the backed Pods.

The case of Docker Compose is similar, since all containers are attached to the same network and can connect with each other without restrictions. The discovery problem is solved by using aliases in service definitions, and the DNS name of the receiver component resolves to the IP addresses of its running containers.

- **Native connectors**.

Native connectors are not specified by the user, they are directly supplied by the underlying platform instead. Therefore each target platform may implement different native connectors using different artifacts. For the sake of example, the Deployer tool recognizes the LoadBalancer native connector (see connector *lb* in Fig. 8). Let us see how this concept is managed in the translation process.

In Kubernetes, the LoadBalancer connector gets translated to a Service which backs all the Pods of the receiver component. The service is registered in the DNS server with the name of the connector and therefore that name resolves to the Service fixed address. When the Service receives incoming connections they are load balanced among the backed Pods.

Docker Compose does not provide any load balancing artifact out of the box so the Deployer tool needs to provide an ad hoc implementation in the translation process. To that end, a service with a single container including the open source HAProxy[6] software is defined and correctly configured to load balance the incoming connections among the replicas of the receiver component. To obtain the replicas of the receiver component, the "standard" mechanism of obtaining first the name of the receiver component (through an environment variable) and resolving it later using a DNS query is used. So, in a sense, the native LoadBalancer connector is internally implemented through a user-defined connector, as explained next.

- **User-defined connectors**.

These connectors are implemented by user-defined components, which are just regular components providing an in endpoint and an out endpoint, and internally "make their connectivity magic happen". To simplify the discussion let us consider user-defined connectors backed by basic components. In this case, all assumptions about basic components apply. In particular, the code implementing the connector can trust to find the DNS name of the receiver component in the *XX_DNS* environment variable and can resolve it to the actual IP addresses of the receiver peers. There is, however, an important difference between a connector and a component: the connector must be always available under a fixed IP address which hides all the complexity happening behind it.

In Kubernetes, user-defined connectors backed by basic components are translated to basic components, but the up-front Service is not headless, but a regular Service instead, enabling load balancing capabilities and publishing a single IP address. This is the case of connector *c* in Fig. 8.

In Docker Compose, a slightly different strategy is followed. First the service implementing the basic component is generated. This service cannot receive the name of the connector though, since its replicas must be hidden behind a single IP address. Therefore, the name of the service is modified, appending the *-impl* suffix. Then an additional up-front native LoadBalancer is generated, receiving the name of the connector and hiding all the replicas of the basic component under a single IP address. Fig. 8 shows how the *c* connector gets translated to the *c* service, implemented internally as a load balancer as explained above, and the *c-impl* service, containing all the replicas of the connector user-defined implementation.

- **Composites**.

Finally, all the previous discussion centers around basic components. Things become a little bit blurrier when considering composites on different nesting levels. To put it simple, a composite can be considered as a piece of system in which all the inputs lead to connectors. This constraint contributes to see a composite instance as a single entity with a single address from any potential sender, and simplifies reasoning about how multiple composite instances get connected in complex architectures.

Having said that, the modeling language proposed by this work enables the definition of dynamically scalable composites, whose number of instances may increase or decrease according to some policies. However, the targeted platforms do not provide native artifacts for doing that, so additional controllers should be devised. This is one known limitation of the Deployer experimental tool, it does only allow the translation and deployment of singleton composites.

7.3. Model deployment

The output of the translation phase is a folder containing a bunch of files describing the artifacts required for deploying the model to a particular target platform. The next step involves deploying all those artifacts in a safe way. Two main challenges should be addressed here, as described in the following paragraphs.

- **Artifacts grouping**.

A big microservice-based model may contain many components. Translating and deploying such a model may involve the generation and deployment of a great deal of artifacts to the target platform. For management purposes, there should be a simple and effective mechanism for identifying and managing all these artifacts and eventually undeploying them.

Kubernetes provides a resource called Namespace for grouping resources together. Namespaces allow to logically isolate collections of resources, and to manage them as a single unit, for example for destruction. Using this mechanism it is easy to include every artifact generated for a model deployment within the same namespace.

Docker Compose provides projects as a mechanism for identifying and managing independent deployments. Each project creates an isolated environment with its own network and all created containers are prefixed with the project name and connected to such a network. Then the environment can be managed independently from others.

- **Artifacts dependencies**.

[6] https://www.haproxy.org.

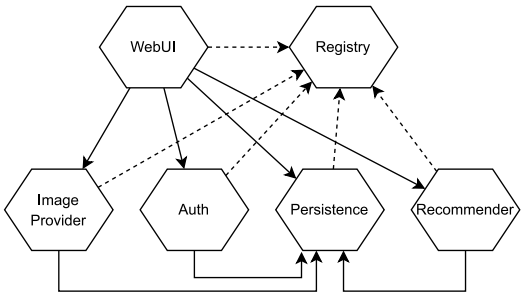

Fig. 9. TeaStore logical architecture from von Kistowski et al. (2018).

In general, artifacts should not be deployed to the target platform in any arbitrary order, due to dependencies. Two main types of dependencies between artifacts can be identified:

1. Dependencies coming from the model: some components require the services of others in order to properly work, dictating a specific order in the initialization process. These dependencies are declared in the model using out endpoints and connectors. For example, in Fig. 8 the component *b1* makes use of the services published by the component *b2*. So it makes sense to create the artifacts required for deploying component *b2* before the artifacts required for component *b1*. A general solution involves inferring a partial order using the topological sorting algorithm and deploying the artifacts following that specific order.
2. Dependencies coming from the platform: depending on the target platform, some artifacts may depend on others, and thus may force to create them in a specific order. For example, in Kubernetes a HorizontalPodAutoscaler refers to an existent ReplicaSet.

In Kubernetes, there are no specific mechanisms for addressing (1). The most common argument in this regard claims that a microservice should not break if the dependent endpoints are not up, and if it breaks, Kubernetes will eventually restart it in subsequent attempts, until the required dependencies are solved. Some workarounds could be enabled though, like using specific scripts checking the availability of the dependent services in the Pods initialization process. Regarding (2) it is necessary to deploy the artifacts in a specific order. To that end, the files generated in the translation phase are prefixed with a sequentially incremented number. Later on, all these files are listed in order and deployed sequentially to Kubernetes.

In Docker Compose, (1) can be addressed using dependencies between services. Compose guarantees to initialize first the dependencies and later the dependent services. And (2) is not a problem since a single root specification file including all the others is generated.

8. Evaluation

This section evaluates the presented work in order to demonstrate (i) that it is applicable to most microservice-based applications and (ii) it fulfills the requirements presented in Section 2.

Regarding (i), to the best of the authors' knowledge there does not exist a database of ready-to-use microservice-based applications to check the modeling language against to. However, it is widely known that best practices recommend to apply popular patterns as building blocks (Taibi et al., 2018; Vale et al., 2022). Then the approach taken by this work is to provide a catalog of sample models describing most of these patterns. For space reason it is not possible to describe them here, but they are publicly available[7] to the interested reader.

[7] https://github.com/jaespei/komponents/blob/main/samples/patterns.

Regarding (ii) the evaluation will be conducted by applying the modeling language in the context of a realistic microservice-based application. To that end, a reference microservice-based application called TeaStore (von Kistowski et al., 2018) has been selected. The TeaStore is an online store for tea and tea related utilities. An implementation of the application is available in a GitHub repository[8] and its general architecture is presented in Fig. 9. The application contains five microservices and a service registry. The service registry is used by the other microservices to register and look for each other (represented with dashed line arrows). The Persistence microservice is consumed by four microservices, two of them consume it only at startup (the Image Provider and the Recommender), and the other two consume it on a regular basis. It is easy to see that even though this simple architecture model provides a basic foundation about the existent components and their connections, it also presents some shortcomings:

- It is an inaccurate version of the real architecture. For example, the Persistence microservice needs to access a relational database backend, which is omitted in the model. There are no details about microservice interfaces, microservice scalability, etc. Consulting the application implementation is required for obtaining all these details.
- The model integrates the registry pattern for solving the service discovery problem. This concrete solution conditions and entangles the application architecture, adding additional dependencies between microservices. These dependencies are artificial because they are not derived from the application domain, but from infrastructure concerns instead, and these concerns might be solved using different mechanisms.
- The model does not reveal how the instances of a microservice communicate with the instances of another microservice. According to the application implementation, microservices make use of Netflix's ribbon[9] software for load-balancing requests among the targets. This circumstance leads to the following insights: on the one hand, requests are using the load-balancer communication pattern, but they might use other patterns like pub-sub or master–slave, and this fact is not visible in the diagram; on the other hand, the application makes use of the client-side discovery pattern, which requires embedding additional communications-aware code into every microservice, decreasing microservice cohesion.

For all these reasons a new attempt to describe the TeaStore application architecture using the modeling language proposed in this paper is presented in Fig. 10. In the authors' opinion this model describes more accurately the application architecture. Both the original and the refined architecture models are publicly available[10] and can be easily deployed using the Deployer tool (see Section 7).

In the model describing the original architecture (see Fig. 9), the registry is incarnated by a regular component and the remainder components connect to it through links in order to register and solve their dependencies. This model can be deployed using the Deployer tool without modifying the source code of the TeaStore reference application.

In addition, the refined model (see Fig. 10) captures the real microservice dependencies, which are connected through load balancer connectors. It also removes infrastructure-related details, like the registry pattern for solving the service discovery problem. To deploy this model using the Deployer tool the source code of the application requires some minimal changes though. Specifically, three files related to solve references between services have been modified.

[8] https://github.com/DescartesResearch/TeaStore.
[9] https://github.com/Netflix/ribbon.
[10] https://github.com/jaespei/komponents/blob/main/samples/teastore/README.md.

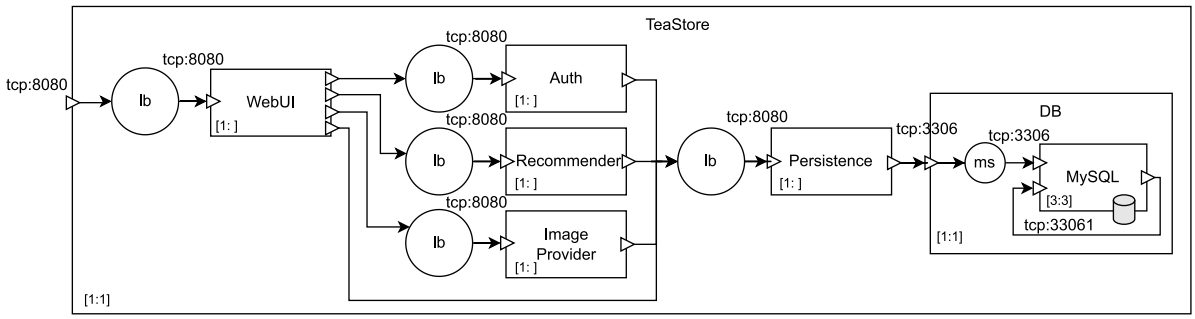

Fig. 10. TeaStore refined model using the proposed modeling language.

In the remainder of this section the refined model is taken as a test bench, and the modeling language is evaluated against the requirements presented in Section 2. To conduct such evaluation, the indicators presented in Fig. 1 are checked.

- **R1: Platform agnostic**.

The modeling language is clearly platform agnostic. On the one hand, it manages abstract concepts not bound to any concrete platform. On the other hand, these concepts (components and connectors) have revolved around software architects and developers for a long time, so they are very familiar to them. Finally, they are easily translatable to modern container-based frameworks, as most of them naturally match their native artifacts.

These claims can be easily checked by deploying the same model to both Kubernetes and Docker Compose using the Deployer tool. In each case, the most appropriate artifacts are generated, as described in Section 7.

- **R2: Components**.

The modeling language provides enough tools to properly characterize individual components. Interfaces with published and required endpoints can be clearly defined. The computing resources required by a component as well as its persistent storage can be easily defined through string-based properties and volumes respectively. Finally, component life-cycle can be managed by defining event handlers. All these aspects are properly identified in the language abstract syntax presented in Fig. 2.

Taking as example the model in Fig. 10, the microservice WebUI publishes an endpoint on port 8080 and presents four dependencies with the other components in the application. In the diagram, the required computing resources and event handlers are missing, but they would be certainly present in the model specification. Finally, the MySQL component requires a volume for persistent storage.

- **R3: Scalability**.

Scalability aspects are properly captured in the language. On the one hand, the language enables the definition of component multiplicities using ranges. In the model presented in Fig. 10 most microservices require at least one running instance, but have no upper bound. The composite DB and the MySQL component require exactly one and three running instances respectively.

On the other hand, elasticity rules which dictate how replicas get created and destroyed are defined using specific policies. The model presented in Fig. 10 does not include these policies but they should be included in the complete specification, as defined in the language abstract syntax in Fig. 2.

- **R4: Connectors**.

The language clearly considers connectors as first-class citizens, and permits the definition of any kind of connector with the ability of taking additional responsibilities related to security, fault tolerance, traceability or any other emergent property.

In the diagram presented in Fig. 10 three different connector types have been identified: load-balancer connectors in front of every dynamically scalable component, a master–slave connector in front of the MySQL component and a full connector in front of the composite DB. Each connector will implement a different behavior, potentially acquiring additional responsibilities. For example, in the event of failures, the master–slave may redirect requests to a different peer in round-robin, until one answers, which is considered the master.

- **R5: Aggregation**.

The language supports aggregation, and hierarchical layouts are allowed through the use of composite components. Composite expansion and appropriate matching between the composite external interface and internal subcomponents can be easily checked by using the formal constructs presented in Section 5.

In the diagram presented in Fig. 10 the application itself is modeled as a composite which just exports one endpoint and gets internally decomposed into multiple interconnected subcomponents. In addition, the DB subcomponent is also modeled as a composite which houses a reliable MySQL 3-node master–slave cluster.

- **R6: Backed by a sound formalism**.

Language syntax and semantics have been formally presented in Sections 4 and 5 respectively. The syntax has been presented using a well-known mechanism among DSL designers, the OMG Meta Object Facility (MOF), which is an easy to interpret class diagram, whereas semantics are based on a (rather sophisticated) variant of the concept of graph, so familiar to software architects. The mapping of syntactic constructs to semantic objects is very intuitive, which will undoubtedly simplify the understanding and adoption of the language, achieving the desired compromise between rigor and ease of use.

- **R7: Pragmatic and easy to use**.

The language is easily applicable to real use cases, as this paper demonstrates with the TeaStore reference application. One of the keys of this applicability and ease of use involves managing a few familiar and simple concepts, but powerful enough for describing complex architectures. In addition, a tool has been implemented to simplify model creation (see Section 6). Also, designing new tools for translating models to concrete platforms is not a difficult task either, and a Deployer tool (see Section 7) has been implemented in order to prove this.

9. Conclusions

Even though much work has been done in the arena of architectural description languages, today there are still many deficiencies in most of the languages proposed in the literature that hinder their use with real applications running in real platforms, for example with microservice-based applications running in modern container-based platforms.

In order to find an appropriate language for modeling microservices, this paper first identifies common concerns found in microservice architectures. Then it presents in a reasoned way the features required by a suitable language for facing most of these concerns. These features are evaluated on the different waves of architectural description languages. Due to the difficulties of finding a language meeting all the requirements, a new language is presented. The language mainly focuses on describing scalable architectures, where multiple replicas of each component run independently.

First, the abstract syntax of the language is described using MOF, along with the conceptual framework that such language imposes. Next a generic mechanism for mapping it to different serialization languages, like YAML, is proposed.

Then the semantics of the language are addressed. To that end, hypergraphs are set as the semantic domain. A brief review of hypergraph-based work in the field is conducted and an alternative methodology for modeling scalable architectures is presented. This procedure is based on defining an architectural style by means of a hierarchical type hypergraph. Finally, an algorithm is presented for obtaining said hierarchical type hypergraph from any specification using the proposed modeling language.

To validate the approach, a couple of tools have been developed, a modeler and a deployer. The first one allows to create models in a graphical way whereas the second one is able to translate and deploy those models to two popular container orchestrators Kubernetes and Docker Compose.

Finally, the language is evaluated in order to check whether it fulfills the proposed requirements. The evaluation of the language is supported by (i) providing a set of sample models describing microservice patterns and (ii) by modeling a well-known reference application.

CRediT authorship contribution statement

Javier Esparza-Peidro: Writing – review & editing, Writing – original draft, Validation, Software, Methodology, Investigation, Funding acquisition, Formal analysis, Conceptualization. **Francesc D. Muñoz-Escoí:** Writing – review & editing, Validation, Supervision, Formal analysis, Conceptualization. **José M. Bernabéu-Aubán:** Supervision, Conceptualization.

Declaration of competing interest

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

Data availability

No data was used for the research described in the article.

References

Abdollahi Vayghan, L., Saied, M.A., Toeroe, M., Khendek, F., 2018. Deploying microservice based applications with Kubernetes: Experiments and lessons learned. In: 11th Int. Conf. on Cloud Comput. (CLOUD). IEEE, pp. 970–973. http://dx.doi.org/10.1109/CLOUD.2018.00148.

Allen, R.J., 1997. A Formal Approach to Software Architecture (Ph.D. thesis). Carnegie Mellon University, Pittsburgh, PA, USA.

Alshuqayran, N., Ali, N., Evans, R., 2016. A systematic mapping study in microservice archit. In: 9th Int. Conf. on Serv.-Oriented Comput. and Appl. (SOCA). IEEE, pp. 44–51. http://dx.doi.org/10.1109/SOCA.2016.15.

Barišić, A., Amaral, V., Goulão, M., Barroca, B., 2011. Quality in use of domain-specific languages: A case study. In: 3rd ACM SIGPLAN Workshop on Eval. and Usability of Program. Lang. and Tools. PLATEAU '11, ACM, New York, NY, USA, pp. 65–72. http://dx.doi.org/10.1145/2089155.2089170.

Bergmayr, A., Breitenbücher, U., Ferry, N., Rossini, A., Solberg, A., Wimmer, M., Kappel, G., Leymann, F., 2018. A systematic review of cloud modeling languages. ACM Comput. Surv. 51 (1), 22:1–22:38. http://dx.doi.org/10.1145/3150227.

Bernstein, D., 2014. Containers and cloud: From LXC to Docker to Kubernetes. IEEE Cloud Comput. 1 (3), 81–84. http://dx.doi.org/10.1109/MCC.2014.51.

Bradbury, J.S., Cordy, J.R., Dingel, J., Wermelinger, M., 2004. A survey of self-management in dynamic software architecture specifications. In: 1st ACM SIGSOFT Workshop on Self-Manag. Syst. (WOSS). ACM, New York, NY, USA, pp. 28–33. http://dx.doi.org/10.1145/1075405.1075411.

Brogi, A., Neri, D., Soldani, J., 2020. Freshening the air in microservices: Resolving architectural smells via refactoring. In: Serv.-Oriented Comput. Workshops (ICSOC). Springer, pp. 17–29. http://dx.doi.org/10.1007/978-3-030-45989-5_2.

Brogi, A., Soldani, J., 2013. Matching cloud services with TOSCA. In: Adv. in Serv.-Oriented and Cloud Comput.. Springer, pp. 218–232.

Bruni, R., Bucchiarone, A., Gnesi, S., Melgratti, H., 2008a. Modelling dynamic software architectures using typed graph grammars. Electron. Notes Theor. Comput. Sci. 213 (1), 39–53.

Bruni, R., Lluch-Lafuente, A., Montanari, U., 2008b. Style-based architectural reconfigurations. Bull. EATCS 94, 161–180.

Bryant, B., Gray, J., Mernik, M., Clarke, P., France, R., Karsai, G., 2011. Challenges and directions in formalizing the semantics of modeling languages. Comput. Sci. Inf. Syst. 8, http://dx.doi.org/10.2298/CSIS110114012B.

Canonical Ltd, 2012. Juju. https://juju.is/, Accessed: 2024-02-02.

Chen, K., Sztipanovits, J., Neema, S., 2005. Toward a semantic anchoring infrastructure for domain-specific modeling languages. In: 5th ACM Int. Conf. on Embed. Softw. (EMSOFT). ACM, New York, NY, USA, pp. 35–43. http://dx.doi.org/10.1145/1086228.1086236.

Courcelle, B., Engelfriet, J., Rozenberg, G., 1993. Handle-rewriting hypergraph grammars. J. Comput. Syst. Sci. 46 (2), 218–270.

van Deursen, A., Klint, P., Visser, J., 2000. Domain-specific languages: An annotated bibliography. SIGPLAN Not. 35 (6), 26–36. http://dx.doi.org/10.1145/352029.352035.

Di Francesco, P., Lago, P., Malavolta, I., 2019. Architecting with microservices: A systematic mapping study. J. Syst. Softw. 150, 77–97. http://dx.doi.org/10.1016/j.jss.2019.01.001.

Drewes, F., Hoffmann, B., Plump, D., 2002. Hierarchical graph transformation. J. Comput. System Sci. 64 (2), 249–283. http://dx.doi.org/10.1006/jcss.2001.1790.

Drewes, F., Kreowski, H., Habel, A., 1997. Hyperedge replacement, graph grammars. In: Rozenberg, G. (Ed.), Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations. World Scientific, pp. 95–162. http://dx.doi.org/10.1142/9789812384720_0002.

ECMA Int, 2017. ECMA-404: The JSON data interchange syntax. URL https://ecma-Int..org/publications-and-standards/standards/ecma-404/, Accessed: 2024-02-02.

Ehrig, H., Ehrig, K., Prange, U., Taentzer, G., 2006. Fundamentals of Algebraic Graph Transformation. In: (Monographs in Theoretical Computer Science), Springer, Berlin, Heidelberg.

Ehrig, H., Pfender, M., Schneider, H.J., 1973. Graph-grammars: An algebraic approach. In: 14th Annual Symp. on Switch. and Automata Theory (SWAT). pp. 167–180. http://dx.doi.org/10.1109/SWAT.1973.11.

Erl, T., 2005. Service-Oriented Architecture: Concepts, Technology, and Design. Prentice Hall PTR, USA.

Fenn, J., Time, M., 2007. Understanding Gartner's hype cycles, 2007. Gartner ID G 144727.

Fowler, M., 2014. Microservices. URL https://martinfowler.com/articles/microservices.html, Accessed: 2024-02-02.

Garlan, D., 2014. Software architecture: A travelogue. In: Workshop on the Futur. of Softw. Eng. (FOSE). ACM, New York, NY, USA, pp. 29–39. http://dx.doi.org/10.1145/2593882.2593886.

Garlan, D., Monroe, R., Wile, D., 1997. ACME: An architecture description interchange language. In: Conf. of the Centre for Adv. Studies on Collab. Res.. CASCON '97, IBM Press, Toronto, Ontario, Canada, p. 7.

Garlan, D., Monroe, R., Wile, D., 2010. Acme: An architecture description interchange language. In: CASCON First Decade High Impact Papers. CASCON '10, IBM Corp., USA, pp. 159–173. http://dx.doi.org/10.1145/1925805.1925814.

Garlan, D., Shaw, M., 1993. An Introduction to Software Architectures. In: Series on Softw. Eng. and Knowl. Eng., vol. 2, World Scientific, pp. 1–39. http://dx.doi.org/10.1142/9789812798039_0001.

Habel, A., 1992. Hyperedge Replacement: Grammars and Languages. In: Lect. Notes Comput. Sci., vol. 643, Springer.

Harel, D., Rumpe, B., 2004. Meaningful modeling: what's the semantics of "semantics"? Computer 37 (10), 64–72. http://dx.doi.org/10.1109/MC.2004.172.

Herbst, N.R., Kounev, S., Reussner, R., 2013. Elasticity in cloud computing: What it is, and what it is not. In: 10th Int. Conf. on Auton. Comput. (ICAC). USENIX Association, San Jose, CA, pp. 23–27.

Hernández-Aparicio, C.C., Ocharán-Hernández, J.O., Cortes-Verdin, K., Arenas-Valdés, M.A., 2022. Architectural languages for the microservices architecture: A systematic mapping study. In: 2022 10th Int. Conf. in Softw. Eng. Res. and Innov. (CONISOFT). pp. 192–201. http://dx.doi.org/10.1109/CONISOFT55708.2022.00033.

Hirsch, D.F., 2003. Graph Transformation Models for Software Architecture Styles (Ph.D. thesis). Facultad de Ciencias Exactas y Naturales, Universidad de Buenos Aires, Argentina.

Hirsch, D., Inverardi, P., Montanari, U., 1998. Graph grammars and constraint solving for software architecture styles. In: 3rd Int. Workshop on Softw. Archit. (ISAW). ACM, pp. 69–72. http://dx.doi.org/10.1145/288408.288426.

Jaramillo, D., Nguyen, D.V., Smart, R., 2016. Leveraging microservices architecture by using Docker technology. In: SoutheastCon 2016. pp. 1–5. http://dx.doi.org/10.1109/SECON.2016.7506647.

JSON Schema, 2020. JSON Schema specification. URL https://json-schema.org/, Accessed: 2024-02-02.

Karabey Aksakalli, I., Çelik, T., Can, A.B., Teki̇nerdoğan, B., 2021. Deployment and communication patterns in microservice architectures: A systematic literature review. J. Syst. Softw. 180, 111014. http://dx.doi.org/10.1016/j.jss.2021.111014.

Kelly, S., Tolvanen, J.-P., 2008. Domain-Specific Modeling: Enabling Full Code Generation. IEEE, Hoboken, N.J.

Kosar, 2010. Comparing general-purpose and domain-specific languages: An empirical study. Comput. Sci. Inf. Syst. 7 (2).

Kratzke, N., Quint, P.-C., 2017. Understanding cloud-native applications after 10 years of cloud computing - A systematic mapping study. J. Syst. Softw. 126, 1–16. http://dx.doi.org/10.1016/j.jss.2017.01.001.

Kumori Systems SL, 2020. Kumori Platform. https://docs.kumori.systems/kpaas/1.0.0/index.html, Accessed: 2024-02-02.

Le Metayer, D., 1998. Describing software architecture styles using graph grammars. IEEE Trans. Softw. Eng. 24 (7), 521–533. http://dx.doi.org/10.1109/32.708567.

Lelovic, L., Mathews, M., Elsayed, A., Cerny, T., Frajtak, K., Tisnovsky, P., Taibi, D., 2022. Architectural languages in the microservice era: A systematic mapping study. In: Conf. on Res. in Adapt. and Convergent Syst. (RACS). ACM, New York, NY, USA, pp. 39–46. http://dx.doi.org/10.1145/3538641.3561486.

Lipton, P., Palma, D., Rutkowski, M., Tamburri, D.A., 2018. TOSCA solves big problems in the cloud and beyond!. IEEE Cloud Comput. 1. http://dx.doi.org/10.1109/MCC.2018.111121612.

Lorido-Botran, T., Miguel-Alonso, J., Lozano, J.A., 2014. A review of auto-scaling techniques for elastic applications in cloud environments. J. Grid Comput. 12 (4), 559–592. http://dx.doi.org/10.1007/s10723-014-9314-7.

Malavolta, I., Lago, P., Muccini, H., Pelliccione, P., Tang, A., 2013. What industry needs from architectural languages: A survey. IEEE Trans. Softw. Eng. 39 (6), 869–891. http://dx.doi.org/10.1109/TSE.2012.74.

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

Mernik, M., Heering, J., Sloane, A.M., 2005. When and how to develop domain-specific languages. ACM Comput. Surv. 37 (4), 316–344. http://dx.doi.org/10.1145/1118890.1118892.

Montesi, F., Weber, J., 2016. Circuit breakers, discovery, and API gateways in microservices. URL https://arxiv.org/abs/1609.05830, arXiv preprint arXiv:1609.05830, Accessed: 2024-02-02.

Newman, S., 2015. Building Microservices : Designing Fine-Grained Systems. O'Reilly Media, Incorporated.

OASIS, 2013a. Topology and orchestration specification for cloud applications (TOSCA) primer version 1.0. URL http://docs.oasis-open.org/tosca/tosca-primer/v1.0/tosca-primer-v1.0.pdf, Accessed: 2024-02-02.

OASIS, 2013b. Topology and orchestration specification for cloud applications version 1.0. URL http://docs.oasis-open.org/tosca/TOSCA/v1.0/TOSCA-v1.0.html, Accessed: 2024-02-02.

Object Management Group, 2012. About the service oriented architecture modeling language specification version 1.0.1. URL https://www.omg.org/spec/SoaML/About-SoaML/, Accessed: 2024-02-02.

Object Management Group, 2016. About the meta object facility specification version 2.5.1. URL https://www.omg.org/spec/MOF/About-MOF/, Accessed: 2024-02-02.

Object Management Group, 2017. About the unified modeling language specification version 2.5.1. URL https://www.omg.org/spec/UML/About-UML/, Accessed: 2024-02-02.

Opara-Martins, J., Sahandi, R., Tian, F., 2014. Critical review of vendor lock-in and its impact on adoption of cloud computing. In: Int. Conf. on Inf. Society (I-Society). pp. 92–97. http://dx.doi.org/10.1109/i-Society.2014.7009018.

Open SOA, 2007. Service component architecture - SCA. URL https://www.osoa.org/, Accessed: 2019-02-26.

Pahl, C., Brogi, A., Soldani, J., Jamshidi, P., 2019. Cloud container technologies: A state-of-the-art review. IEEE Trans. Cloud Comput. 7 (3), 677–692. http://dx.doi.org/10.1109/TCC.2017.2702586.

Papazoglou, M.P., Traverso, P., Dustdar, S., Leymann, F., 2007. Service-oriented computing: State of the art and research challenges. Computer 40 (11), 38–45. http://dx.doi.org/10.1109/MC.2007.400.

Richards, M., 2015. Microservices Vs. Service-Oriented Architecture. O'Reilly Media.

Schmidt, D., 2006. Guest editor's introduction: Model-driven engineering. Computer 39 (2), 25–31. http://dx.doi.org/10.1109/MC.2006.58.

Slonneger, K., Kurtz, B., 1995. Formal Syntax and Semantics of Programming Languages: A Laboratory Based Approach, first ed. Addison-Wesley Longman Publishing Co., Inc., USA.

Soldani, J., Tamburri, D.A., Van Den Heuvel, W.-J., 2018. The pains and gains of microservices: A systematic grey literature review. J. Syst. Softw. 146, 215–232. http://dx.doi.org/10.1016/j.jss.2018.09.082.

Söylemez, M., Tekinerdogan, B., Kolukısa Tarhan, A., 2022. Challenges and solution directions of microservice architectures: A systematic literature review. Appl. Sci. 12 (11), http://dx.doi.org/10.3390/app12115507.

Taibi, D., Lenarduzzi, V., Pahl, C., 2018. Architectural patterns for microservices: A systematic mapping study. In: 8th Int. Conf. on Cloud Comput. and Serv. Sci. (CLOSER). SciTePress, Funchal, Madeira, Portugal, pp. 221–232.

The Linux Foundation, 2017. HELM: The package manager for kubernetes. https://helm.sh/, Accessed: 2024-02-02.

Tran, V.X., Tsuji, H., 2009. A survey and analysis on semantics in QoS for web services. In: Int. Conf. on Adv. Inf. Netw. and Appl. (AINA). pp. 379–385. http://dx.doi.org/10.1109/AINA.2009.43.

Vale, G., Correia, F.F., Guerra, E.M., de Oliveira Rosa, T., Fritzsch, J., Bogner, J., 2022. Designing microservice systems using patterns: An empirical study on Quality Trade-Offs. In: 19th Int. Conf. on Softw. Archit. (ICSA). IEEE, pp. 69–79. http://dx.doi.org/10.1109/ICSA53651.2022.00015.

Vaquero, L.M., Rodero-Merino, L., Caceres, J., Lindner, M., 2009. A break in the clouds: Towards a cloud definition. SIGCOMM Comput. Commun. Rev. 39 (1), 50–55. http://dx.doi.org/10.1145/1496091.1496100.

von Kistowski, J., Eismann, S., Schmitt, N., Bauer, A., Grohmann, J., Kounev, S., 2018. TeaStore: A micro-service reference application for benchmarking, modeling and resource management research. In: 26th Int. Symp. on Model., Anal., and Simul. of Computer and Telecommun. Syst. (MASCOTS). IEEE, pp. 223–236.

Vural, H., Koyuncu, M., Guney, S., 2017. A systematic literature review on microservices. In: Int. Conf. Comput. Sci. and Its Appl. (ICCSA). Springer, pp. 203–217.

Wile, D., 2004. Lessons learned from real DSL experiments. Sci. Comput. Program. 51 (3), 265–290. http://dx.doi.org/10.1016/j.scico.2003.12.006.

YAML Language Development Team, 2021. YAML specifications. URL https://yaml.org/, Accessed: 2024-02-02.

Zaafouri, K., Chaabane, M., Rodriguez, I.B., 2021. Systematic literature review on service oriented architecture modeling. In: Int. Conf. Comput. Sci. and Its Appl. (ICCSA). Springer, pp. 201–210. http://dx.doi.org/10.1007/978-3-030-86970-0_15.

Javier Esparza-Peidro currently works as an associate professor at UPV where he has taught subjects related to programming, software engineering and distributed systems for twenty years, supervising more than 80 final degree projects. He is a member of the Distributed Systems research group from 2001. His research interests cover distributed systems, cloud computing and big data.

Francesc D. Muñoz-Escoí received a Ph.D. in Computer Science from Universitat Politècnica de València (UPV) in 2001. He currently works as an associate professor at UPV and as a senior researcher at its Instituto Universitario Mixto Tecnológico de Informática. He has published more than 110 papers in international conferences and journals. His research interests cover multiple distributed system areas: distributed algorithms, distributed data management, elastic services, cloud computing, and blockchain systems.

José M. Bernabéu-Aubán obtained his Ph.D. in Computer Science from Georgia Institute of Technology (USA) and currently works as a full professor at UPV where he leads the Distributed Systems research group. Professor Bernabéu has led the Instituto Universitario Mixto Tecnológico de Informática at UPV from 1994 to 2004 and since 2014 up to now. From 2004 to 2011, he joined Microsoft Corp. working actively in the architecture, design and development of the Windows Azure platform, co-authoring some of its patents. He has written multiple papers in journals and conferences in different distributed system areas. He has led more than 30 research projects in those fields.