Received August 27, 2021, accepted September 12, 2021, date of publication September 23, 2021, date of current version October 5, 2021.

*Digital Object Identifier 10.1109/ACCESS.2021.3115438*

# Service Relationship Orchestration: Lessons Learned From Running Large Scale Smart City Platforms on Kubernetes

**MERLIJN SEBRECHTS, SANDER BORNY, TIM WAUTERS, (Member, IEEE), BRUNO VOLCKAERT, (Member, IEEE), AND FILIP DE TURCK, (Fellow, IEEE)**
IDLab, Department of Information Technology, Ghent University - imec, B-9052 Ghent, Belgium

Corresponding author: Merlijn Sebrechts (merlijn.sebrechts@ugent.be)

This work was supported in part by the Project ''SynchroniCity: Delivering an IoT enabled Digital Single Market for Europe and Beyond'' funded by the European Union's Horizon 2020 Research and Innovation Programme under Grant 732240.

**ABSTRACT** Smart cities aim to make urban life more enjoyable and sustainable but their highly heterogeneous and distributed context creates unique operational challenges. In such an environment, multiple companies work together with government on applications and data streams spanning several management domains. Deploying these applications, each of which consists of several connected services, and maintaining an overview of application topologies remains difficult. Even though cloud modelling languages have been proposed to solve similar issues, they are not well fit for such a heterogeneous environment because they often require an ''all or nothing'' approach. Moreover, cloud modelling languages add an additional abstraction layer that rarely supports all features of the underlying platform and make it harder to reuse existing knowledge and tools. This research defines *service relationships* as the key element to modelling applications as topologies of services. We use this definition to pinpoint what is lacking in the state of the art Kubernetes orchestration tools and provide a blueprint for how relationship support can be added to any orchestrator. We present ''orcon'', a proof of concept orchestrator that extends the Kubernetes API to allow managing relationships between services by adding metadata to service definitions. Our evaluation shows this orchestrator enables lifecycle synchronization and configuration change propagation with an overhead of only 0.44 seconds per service.

**INDEX TERMS** Cloud models container orchestration, cross-domain, kubernetes, microservices, service composition, service dependencies, service orchestration, smart cities, topology models, TOSCA.

## I. INTRODUCTION

Smart cities have the potential to make urban life more enjoyable and sustainable by introducing deep integration with Internet of Things (IoT) technology. The inherent heterogeneous and collaborative nature of cities creates unique challenges for integrating IoT. Problems cannot be solved in a vacuum: they often require collaboration between multiple competing industry partners, governments, research institutions and the public. A single end-to-end application can have data streams crossing multiple management domains and environments: it can contain components managed by completely different teams on different networks, clouds and data centers. This unique environment acts as a multiplier to operational complexity, making it hard for developers to focus on the applications themselves.

One such smart city project is ''City of Things'' [1], transforming the city of Antwerp, Belgium, to tackle a wide variety of challenges such as prediction and detection of flooding, improving traffic flows, creating a smart grid and fine-grained monitoring of pollution. Since inter-disciplinary research, citizen science and industry collaboration are key here. The speed of innovation and the exploratory nature of this project only exacerbates the operational challenges, making it hard to get a clear picture of application topologies and how services are connected. Furthermore, due to this nature, individual services are changed often. Adapting related services to these changes often requires manual effort and close

The associate editor coordinating the review of this manuscript and approving it for publication was Taehong Kim.

collaboration between teams. This speeds down the rate of innovation.

Cloud modelling languages such as the OASIS Topology and Orchestration Specification for Cloud Applications (TOSCA) are a great way to model the full topology of an application. Cloud orchestrators can then automatically manage individual services and their relations to each other. However, these languages often follow an all-or-nothing approach in that the entire application needs to be modeled in that language. Moreover, they create an additional abstraction on top of the existing tools and platforms that developers use. Since the goal of the City of Things project is to explore what could be possible in a Smart City context, it is hard to define requirements and commit to specific technology choices early in the process. Container managers such as Kubernetes (k8s) [2] provide very flexible APIs to manage containerized services. However, dependency-management and collaboration between services require ad-hoc solutions that are prone to break. Service meshes can aid in this, but they require putting additional components such as proxies in the data path. This overhead is even more damning if they are only used to solve operations issues. Given the latency requirements of many IoT applications such as smart grids, this is not a good fit for Smart Cities projects. Finally, service meshes provide a general fabric for all services to communicate with each other, instead of configuring specific connections based on a topology model of the application.

Thus, given these limitations of the state of the art, the contribution of this work is to answer the following research questions.

*RQ 1:* On an abstract level, what concepts enable modeling and automated management of dependencies between services?

*RQ 2:* To what extend does the state of the art support modeling and automated management of such dependencies in Kubernetes?

*RQ 3:* How can existing platforms be extended in order to support service relationships without hiding the underlying API of the platform to users and without adding extra components in the data path?

*RQ 4:* What is the orchestration overhead introduced by adding support for such relationships?

Additionally, this research provides an open source proof of concept prototype relations orchestrator for Kubernetes. This paper starts off by exploring how related work addresses some of the operational challenges of running complex interconnected applications and services in Section II. The smart cities use-case is explained in detail in Section III. Section IV provides a number of definitions concerning what it means to have a relationship between two services and Section V uses these definitions to pinpoint what is lacking in the state of the art Kubernetes orchestration tools. We propose ‘‘orcon’’, a proof of concept orchestrator on top of Kubernetes that manages relationships between services in section VI. We evaluate whether this approach is viable by comparing the proof of concept with the Juju orchestrator and native Kubernetes tools in Section VII. Finally, Section IX concludes this paper and explains how future work will continue this line of research.

## II. RELATED WORK

Although cloud models are often presented as the solution to many operational challenges, their creation requires considerable technical and architectural expertise [3]. However, some of this expertise is already being captured in the form of cloud computing patterns. Fehling *et al.* [4] propose a way to describe the deployment in an abstract way using patterns, and to translate these patterns into the actual deployment models, with the goal of significantly reducing the required knowledge to create cloud models. Martino *et al.* use automated reasoning to map between cloud agnostic and vendor dependent cloud patterns [5]. Unfortunately, these approaches are not applicable to scenarios where full access to the underlying cloud infrastructure is required because it is hidden by the abstractions of the structural models. This eliminates the possibility of a multilevel approach where deployments can be modified using both higher-level and lower-level concepts.

Container orchestration is another recent development aiming to solve operational challenges. Topics such as resource scheduling, load balancing, fault tolerance and autoscaling are supported in most state-of-the art orchestrators [6]. However, higher-level abstractions, and specifically, the concept of relationships and dependencies is much less widespread. As Burns *et al.* note, Google's decade of experience with container orchestration has shown that dependency management is an important issue but the perceived complexity of dependency-aware systems has hampered the adoption of such systems by mainstream container-management systems [7].

### A. CLOUD MODELS AS AN ABSTRACTION

Cloud models have also been proposed as a way to decrease the complexity of managing complex cloud applications by using it as a simpler abstract representation. This has had some success in the area of big data processing, for example [8]. Bhattacharjee *et al.* continue on this line of thinking and propose CloudCAMP [9] for domain-specific modelling so that cloud applications and their dependencies can be modeled without the need for domain expertise. The authors show that providing a higher-level abstraction to model cloud applications indeed reduces the complexity and enhances the ease of use. However, it requires pre-made building blocks to provide the higher-level abstractions. TOSCA is a front-runner in the field of cloud modelling languages and is used in many domains to simplify operations, however, because of its versatility and popularity, the risk exists of proliferation of incompatible TOSCA dialects [10].

### B. MUTATING CLOUD MODELS

Finding effective ways to mutate cloud models is important in order to enable customization and day-2 operations such as maintenance and patching of applications managed with models. Managing dependencies, configuring and re-configuring services all require changing and updating cloud models. Palesandro *et al.* propose Mantus [11] as an aspect-oriented approach for modifying TOSCA models. They introduce the TOSCA Manipulation Language as an "XSLT for TOSCA models", making it possible to model day-2 operations as changes to a cloud model. Some of the solutions in this space come from the industry, with Kustomize [12] as a prominent example of a language to modify Kubernetes models. Some tools such as Helm [13] go one step further by adding features such as package management, lifecycle management and dependencies to Kubernetes models. However, these are heavily criticized even within the Kubernetes community for conflating too many concerns in one tool and solving none of the challenges particularly well [14].

### C. RELATIONSHIPS IN MODELS

In order to make it easier to modify and reuse cloud models, the majority of cloud modelling languages supports creating topologies, where a cloud application is composed of a number of self-contained entities connected by relationships [15]. This is particularly relevant in NFV environments. Chaining of heterogeneous functions is important to both NFV and IoT platforms, although it is still an open research challenge in 2019 according to Vaquero *et al.* [16]. The Juju cloud modelling language, for example, is being used to orchestrate 5G Virtual Network Function (VNF) services [17]. These relationships are also useful for more data flow-based workloads [18], [19].

An important advantage of topology-based cloud modelling languages comes from their ability to reuse components by turning a monolithic cloud model into a set of loosely-connected interchangeable components [18], [20], [21]. This also supports enhanced collaboration between multiple parties, for example by function shipping [22] and can even support a "marketplace"-like ecosystem with off-the-shelf components. Furthermore, these topologies can be used in order to analyze application topologies, find common microservice architectural smells, and suggest refactorings, as shown by Brogi *et al.* [23].

### D. SMART CITY SERVICE ORCHESTRATION

Service orchestration in smart cities is a complex multi-faceted issue. Sivrikaya *et al.* tackle cross-domain service composition by proposing the ISCO framework [24], a multi-agent-based middleware framework, which builds on top of the JIAC agent platform [25]. Not addressed by ISCO, however, is the issue of composition of existing polyglot services running in container orchestration platforms such as Kubernetes. The bIoTope project aims to build an ecosystem to create ad hoc and loosely coupled information flows in a smart cities context [26]. It introduces several building blocks in order to enable standards-based open communication and proof of concept implementations of this framework as an alternative to the traditionally proprietary and vertically-oriented ecosystems. The lower-level concerns of modeling, deployment and reconfiguration of containerized services based on compositions are not in the scope of the bIoTope project, however. The SWITCH workbench [27] offers a solution for managing the entire life cycle of time-critical applications in general. Using TOSCA as a modeling language, it supports management of applications consisting of complex topologies of microservices. Although it supports deploying applications to Kubernetes, it hides the entire Kubernetes API behind the TOSCA abstractions, making it hard to integrate with the wider Kubernetes ecosystem.

## III. THE USE-CASE

As explained in the introduction, the City of Things project [1] is a collaboration between industry, academia and government with the goal of using IoT to make urban life more enjoyable and sustainable. As with any smart cities project, this creates a complex environment spanning multiple management domains that has to support multiple tenants with varying levels of collaboration between them. At the core of this setup, shown in Figure 1, sits Obelisk [28], [29]; a platform for building scalable applications on IoT centric time series data. Obelisk is specifically built to support the heterogeneous nature of smart cities. Heterogeneity in protocols and sensors is supported by using a flexible REST interface and an integration layer capable of translating a wide variety of IoT protocols. Smart cities also introduce a second form of heterogeneity however, namely in terms of authorization, data access and data ownership. Since smart cities require collaboration between multiple parties who are direct competitors to each other, there are very stringent requirements on which data gets shared to which exact parties. Obelisk supports this using deeply integrated multi-tenant isolation with granular access controls in the entire architecture. The Obelisk and City of Things projects are explained in much more detail in previous work [1], [28].

It became clear early in the project that there is a need for low-latency processing and transformation of the data captured by Obelisk. For this reason, the solution includes a multi-tenant Kubernetes-based Platform as a Service (PaaS) co-located with the Obelisk core. This allows customers to run event-based containerized applications that ingest event-based data streams from Obelisk using a Server-Sent Events (SSE) API, process them, and load them into either Obelisk or external platforms. Model-based management of these applications and their connections is the main focus of this paper. Figure 2 shows a simplified and zoomed-in view of the use-case where a number of different applications running inside of the Kubernetes PaaS connect to the SSE server. The *core team* develops and manages the Obelisk core, a number of different *app teams* develop applications using Obelisk's SSE API, and the *platform team* manages the Kubernetes cluster where the applications run. None of these parties has full knowledge and access to all the parts of the entire end-to-end application.

Automated model-based deployment and management of the applications is key here for a few reasons. First of all, it allows app teams to get started with the data as soon as possible and in an independent manner. Secondly, if the automation system propagates changes without human interaction, then all teams have the freedom to modify and iterate on their part of the software without coordination with other teams. Finally, because of the complexity of interconnections between different components, model-based management is key for its ability to retain a global view of the complete topology. This requirement is partly fulfilled by Kubernetes itself. It falls short, however, in modeling and managing the relationships between individual components. For example, it is not possible for Kubernetes to define that one service has a relationship to another, automatically configure the services depending on that relationship, and reconfigure the services when the service on the other end of the relationship changes. Moreover, the heterogeneity of stakeholders in a smart cities context adds additional challenges to service orchestration. First of all, it is very hard to standardize on a single methodology and toolset to deploy and manage applications. Although each application has components running in Kubernetes, these are often deployed and managed differently, depending on the stakeholder. Secondly, it is not possible to have a single stakeholder deploy and manage the entire end-to-end application as a single model because each application crosses multiple management domains.

Due to the technically challenging nature of this project, developers often use advanced features of Kubernetes in order to fine-tune how the applications are managed, scaled and upgraded. This includes integration with many tools in the Kubernetes ecosystem such as service meshes, custom resources and operators. Many of these tools either extend the Kubernetes API with new functionality or use the Kubernetes API to manage and update the application models. As such, using these tools requires direct access to the Kubernetes API and the model of the application. It is thus vital that any solution does not impede the developer's access to the API so they can continue benefiting from the wider Kubernetes ecosystem. Similarly, the solution itself should also integrate into Kubernetes itself in such a way that existing management tools and workflows seamlessly integrate, showing the need for a Kubernetes-native solution.

## IV. CONCEPTS OF A SERVICE RELATIONSHIP

This section provides a definition and an extensive explanation of a service relationship and related concepts. The purpose of this is two-fold. Firstly, this explanation is used throughout this research to evaluate the state of the art. Secondly, this chapter forms a blueprint for how to fully support service relationships in orchestration systems, answering RQ 1. The section starts with the definition and proceeds to explain each part of the definition in detail. The terms in **bold** are used further in this work to refer back to specific parts of the definition.

*Definition 1:* A *service relationship* is an explicit typed connection between isolated and independent service models that enables exchange of configuration information, synchronization of lifecycles and runtime communication.

In the most simple sense, a relationship means that two services are connected with each other. This connection can have up to three distinct components.

1) The **communication component** refers to the interaction of the services at run-time. *Example: The SSE client communicates with the server using the HTTP SSE protocol.*
2) The **lifecycle component** refers to how the lifecycles of related services are dependent on each other. *Example: The SSE client can only start after the SSE server has started.*
3) The **configuration component** refers to how the configuration of one service uses information from another service. *Example: the SSE client app is configured with the URL of the SSE server.*

Each relationship has one or more of these components. The relationship between a web service using an SSL certificate and the certificate authority (CA), for example, has a lifecycle and a configuration component: the web service can only start after the CA is available and the web service is configured to use a certificate signed by the CA. This relationship does not have a communication component, however, since there is no communication between the webserver and the CA at runtime.

Relationships are **explicit** in the sense that they are defined in the model, rather than inferred from service configuration or runtime behavior. This important property makes sure operators have a clear view on the topology of their application, and automation tools have a straightforward way to reason about it.

*Definition 2:* An *interface* is the directional **type** of a service relationship, describing the supported exchange of information and coordination between services.

The interface specifies which components a relationship has and what each component entails. It defines the following.

- How *the lifecycles* of the two services are connected.
- *The protocol* used for runtime communication between the services.
- *What* configuration information is *shared* between the services.
- *How* configuration information is *presented* to each service.

An interface is directional in the sense that the service on each end of a relationship acts differently. As a result of this, an interface can be broken down into two sides: one for each service.

The type of a relationship is defined in an interface: it specifies which components a relationship has and what each component entails. It defines the following.

*Definition 3:* A *role* is the part of an interface describing the supported relationship behavior of a single service.

**TABLE 1. Comparison of relationship support in Kubernetes solutions. Although TOSCA-based solutions like Juju provide the full functionality of service relationships, they fall short in allowing users access to the full functionality of Kubernetes.**

| | K8s | Helm | Service Mesh | Juju |
|---|---|---|---|---|
| **Relationship Support** | | | | |
| Explicit | × | ✓ | × | ✓ |
| Typed | × | × | × | ✓ |
| Isolated | × | × | × | ✓ |
| Independent | × | × | × | ✓ |
| Communication comp. | ✓ | ✓ | ✓ | ✓ |
| Lifecycle comp. | × | × | × | ✓ |
| Configuration comp. | × | ✓ | × | ✓ |
| **Kubernetes Nativity** | | | | |
| K8s API access | ✓ | ✓ | ✓ | × |
| Part of K8s API | ✓ | × | ✓ | × |

Two roles exist in each relationship: the *provider* and the *consumer*. One service *provides* the interface while the other service *consumes* it. A relationship is only possible between a service that supports the *provides* role and one that supports the *consumes* role of the same interface.

Relationships are created between **isolated** services in the sense that model and state are service-scoped. By default, information in the service scope is not available to other services and cannot be referenced in their models. Information can only be made available to related services by including it in a role. This behavior ensures all service relationships are explicit and completely described by the interface.

Furthermore, each related service is **independent**, meaning each service model can be managed as an independent entity, by an independent entity. This makes it possible for service relationships to cross administrative boundaries forming the basis for collaboration between independent parties.

## V. RELATIONSHIP SUPPORT IN KUBERNETES

This section evaluates the current support for relationships in Kubernetes based on the definitions of the previous section. Table 1 shows a summary of this evaluation, presenting the answer to RQ 2.

### A. NATIVE KUBERNETES

In Kubernetes, the desired state of a cloud application is modeled using *object specs*. Although it is possible for two services to communicate with each other, these connections are not explicit in the object specs. The object spec specifies a unique name for each service. All containers in the same namespace as the service can use this name to establish runtime communication. In order to know which services will actually establish this communication, an operator would have to trace network traffic and/or inspect the code and declarations of every single service to see which ones initiate communication, as proposed by Muntoni *et al.* [30]. Although this functionality enables the communication component of a relationship, it does not support a lifecycle component nor a configuration component. Moreover, relationships are not explicit in the model and there is no notion of interfaces or relationship scope.

### B. HELM

Helm is a package manager for Kubernetes. The desired state of an application is modeled in a Helm chart; a combination of templated Kubernetes object specs. Helm provides tools to fill in these templates, deploy the resulting objects and manage their lifecycle. Because these templates are based on Kubernetes object specs, Helm users can take full advantage of the Kubernetes API. Despite that, Helm itself is not part of the Kubernetes API. As a result, tools built to manage Kubernetes applications cannot take advantage of Helm.

Helm makes it possible for a chart to explicitly define its dependencies by specifying subcharts. As the name implies, this is an inherently hierarchical relationship requiring the subchart to be deployed as part of its parent, breaking the independence requirement stated in the previous chapter. As a result, this approach requires an operator who has complete authority over the entire model spanning all connected services. This does not allow creating services that span administrative boundaries as explained in Section III.

Even though relationships in Helm are explicit, they are not typed. Although the model explicitly states *which* services are connected, it does not specify *how* they are connected. An operator needs to ''reverse-engineer'' this information by inspecting the template and checking whether services are configured to communicate with each other, and inspecting the services themselves to see if they have a hard-coded connection to another service. Furthermore, Helm dependencies only provide one-directional isolation: a parent chart can override any values of the subchart while a subchart has no access to the parent chart. Lastly, these dependencies do not influence the lifecycles of the services. When Helm deploys a service, it does not wait until the dependencies of that service are running.

### C. SERVICE MESHES

A service mesh is a relatively new concept that seeks to improve communication between services by providing observability, increased security and failure recovery for requests between services. Istio [31], for example, implements a service mesh to connect containers running in Kubernetes. Conceptually speaking, service meshes aim to solve problems of the communication component of relationships, but they themselves cannot cover other aspects of a relationship between services because they provide no way to influence the lifecycle of connected services nor can they change the configuration of different services. Although service meshes can be useful to discover the topology of a microservice application, they infer this from the runtime behavior of services, instead of any explicit relationship definition. Furthermore, it is up to the administrator to make sure that connected services are actually compatible since service meshes do not have a way to declare and check the type of relationships.

A number of distributed tracing and observability solutions, such as Dynatrace [32], exist for Kubernetes. Much like service meshes, these applications make it possible to intelligently infer dependencies from the runtime behavior of microservice applications and present this dependency information to users as a topology model. Much like service meshes, however, these also suffer from the same issues: they provide no way to manage the lifecycle and configuration of services based on a topology model. In a sense, it's the reverse of our goal: instead of changing run-time behavior to conform with a model, it creates a model based on run-time behavior.

Another, more practical issue specific to service meshes is that these are often implemented using sidecar proxies. These add latency and increase the resulting complexity of a deployment. Moreover, these proxies often only support a limited number of communication protocols.

Although service meshes and distributed tracing solutions are a useful development, they do not provide any additional features over Kubernetes in terms of service relationships as defined in Section IV. For this reason, we see them as complementary to service relationships. Thus, the remainder of this research does not regard these as an alternative solutions but evaluates whether different solutions can integrate with them by determining whether the model allows direct access to the Kubernetes API.

### D. TOSCA-RELATED SOLUTIONS

A number of different initiatives are working towards TOSCA-based solutions to manage applications running on top of Kubernetes. Projects with an industry background such as Juju and Cloudify provide full-featured orchestrators with Kubernetes support. This integration also has considerable interest from academia with a number of recent publications such as Chareonsuk *et al.*, proposing a TOSCA to Kubernetes translator [33], Bogo *et al.* introducing a toolchain for deploying multicomponent applications using TOSCA [34], and Borisova *et al.* examining how to adapt TOSCA for Kubernetes deployment [35].

Because of TOSCA's exhaustive support for relationships, all these tools support explicit and typed modelling of communication [36], lifecycle [37] and configuration components of a service relationship. Moreover, individual components in TOSCA are isolated and a number of TOSCA implementations, such as Juju, support creating relationships between completely independent models. As a result, it is possible in Juju to create relationships crossing administrative boundaries, as required by the use-case explained in Section III. Even though most of these TOSCA-based solutions have complete support for service relationships as defined in Section IV, there are two issues that make them unsuitable for our use-case. The first one is that it is often an all-or-nothing approach: taking full advantage of this relationship functionality is only possible if the entire application is modeled using the TOSCA-based platform. But this is not always feasible, as our use-case shows: many collaborators use their own tools and methodologies to manage their infrastructure and are hesitant to change. The second issue is that TOSCA adds an additional abstraction *on top of* Kubernetes, which hides Kubernetes itself. Such abstractions, when done well, can simplify complex platforms but they have the downside that they often do not support all the features of Kubernetes itself and that they make it hard to reuse existing Kubernetes tools and expertise. Next to this, it also makes migrating to the new abstraction more difficult because there is no a clear migration path available and it is difficult to gradually transition to the new abstraction. For the remainder of this paper, we will use Juju as a representative TOSCA-based solution.

## VI. IMPLEMENTATION

This section presents the open source *orcon* orchestrator [38] developed as part of this research. The motivation behind the development is three-fold. First of all, this implementation allows us to check the validity of the concepts and definition of service relationships presented in Section IV and determine whether these are a sufficient answer to RQ 1. Secondly, this implementation shows how to extend an existing platform to support service relationships without hiding the underlying platform API, answering RQ 3. Finally, this implementation is used to answer RQ 4 by evaluating its performance in Section VII.

The orcon orchestrator injects the concepts of *relationships*, *interfaces* and *roles* into the Kubernetes API. By using injection, orcon avoids creating an additional layer of indirection and enables users to keep working with the same tools and techniques they are used to.

Kubernetes works based on the desired state principle: users declaratively describe the desired state of the system by adding and changing *object specs* in the Kubernetes *API server*. Kubernetes services then take appropriate action to get the system into that state and reflect the current state in the *object status*. The object status thus describes what is actually set up in the cluster in order to meet the needs described in the object spec. Orcon allows users to describe desired relationships between objects by adding additional information to the object specs. Two orcon services monitor the API server for these descriptions and take the appropriate actions to establish the desired relationships. Since modifying object specs is already supported by means of the Kubernetes API, all Kubernetes tools that support this API automatically support orcon too.

The next subsection explains how the conceptual model of a relationship from Section IV is implemented in the existing Kubernetes API in order to allow users to describe desired relationships between objects. The remaining subsections explain how the orcon services extend the Kubernetes control plane in order to establish those relationships.

### A. REPRESENTING RELATIONSHIPS, INTERFACES AND ROLES IN KUBERNETES OBJECTS

Orcon adds a number of extensions to the schema of Kubernetes object specs that are directly mapped to the conceptual model of a relationship detailed in Section IV. For these extensions, orcon uses *Annotations*, which allow adding arbitrary complex metadata to any object, and *Custom Resources*, which allows adding new types of objects to the Kubernetes API.

The **roles and interfaces** supported by an object are described using the *orcon.dev/provides* and *orcon.dev/consumes* annotations. Each annotation contains a comma-separated list of interface names of which the object supports that specific role. These annotations are used by orcon in order to type-check relationships and in order to figure out the interface of a relationship between two objects.

The relationships themselves are described by adding the annotation *orcon.dev/relationship* to the *consumer* end of a relationship. It supports a comma-separated list of object names so that a single object can have multiple relationships. Each object name in this list specifies the request for an individual relationship between the object in question and the named object. Note that in the Kubernetes model of "desired state", these annotations denote the *desire* for a relationship between two objects, not necessarily an *established* relationship. Establishing a relationship is only possible if the specifying object supports the *consumer* role of an interface that the named object *provides*.

These three annotations are enough to describe the intent for a basic relationship between a Kubernetes Deployment and a Kubernetes Service. Below is an example of a Deployment consuming the *sse* and *mysql* interface, which has a relationship to an object named *sse-endpoint* and an object named *mysql-db*.

```
kind: Deployment
metadata:
  name: sleep
  annotations:
    orcon.example/consumes: sse,mysql
    orcon.example/relations: sse-endpoint,mysql-db
spec:
  ...
```

In such a basic relationship, the interface is inferred implicitly based on the default orcon relationship template and the name of the interface. The default interface template is as follows.

- The *lifecycles* of the two objects are connected in such a way that the Deployment starts after the service becomes available.
- Orcon assumes both objects use the same *protocol* for communication.
- The providing Service *shares* its URL.
- The service URL is presented to the consuming Deployment as an environment variable with the same name as the interface itself.

Since such an interface is very limited in its usefulness and provides only rudimentary type-checking, orcon allows users to explicitly define interfaces by specifying both roles of an interface using two custom resources: *ProviderConfig* and *ConsumerConfig*.

The **ProviderConfig** explains how to enact the *provider* role of an interface. It defines which values should be

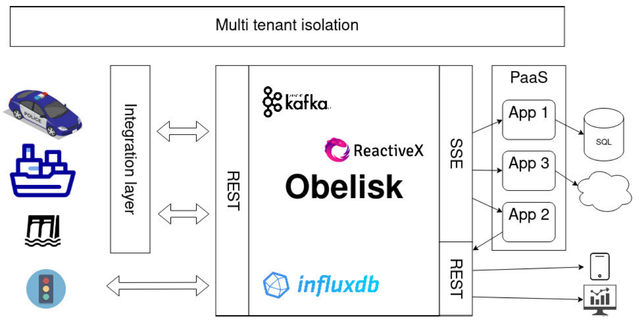


**FIGURE 1. High level overview of the Obelisk city of things architecture. Obelisk provides uniform and secure access to heterogeneous IoT data to multiple tenants with varying levels of cooperation. An in-house PaaS allows tenants to add additional processing functionality close to the data.**

extracted from the providing object. The *valueLocations* map in a ProviderConfig object describes for each relationship key, where to extract the associated value from. These values can come from the object spec, from the object state, a Secret or a ConfigMap.

Below is an example of a ProviderConfig mapping three relationship keys to a field in the object spec and two secrets.

```
apiVersion: relations.orcon.example/v1alpha1
kind: ProviderConfig
metadata:
  name: mysql-config
config:
  valueLocations:
    url:
      type: /v1/services
      name: mysql-service
      path: spec.externalName
    username:
      type: secret
      name: mysql-secret
    password:
      type: secret
      name: mysql-secret
```

Important to note here is that, using this method, a providing service does not actually have to run inside of the Kubernetes cluster itself. The only requirements is that a *representation* of the service is present in the API server in the form of an object. The above example uses the *externalName* functionality of Kubernetes Service objects, which allows representing external services in Kubernetes objects.

The **ConsumerConfig** explains how to enact the *consumer* role of the relationship. It defines how the relationship values should be injected into the consuming object and what kind of lifecycle dependency the consuming object has on the providing object.

- The *lifecycledep* key describes the lifecycle dependency between the provider and the consumer of a relationship. At the moment, the only supported lifecycle dependency is *start*, denoting that the consuming service needs to start *after* the providing service. This field also accepts the string *none*, denoting there is no lifecycle dependency.

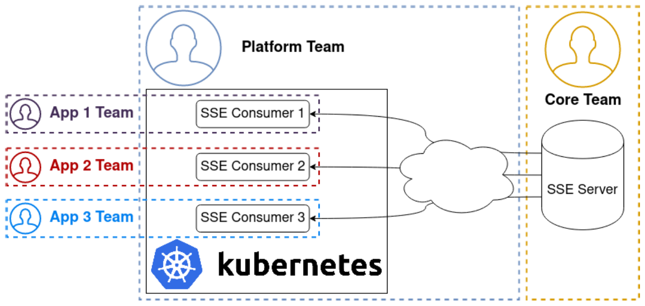


**FIGURE 2.** Different parts of the application run in different administrative domains, shown in the figure using dashed lines. Each third party team is a different tenant on the Kubernetes cluster. The platform team manages the cluster and its connection to external infrastructure. The core team manages an external SSE server.

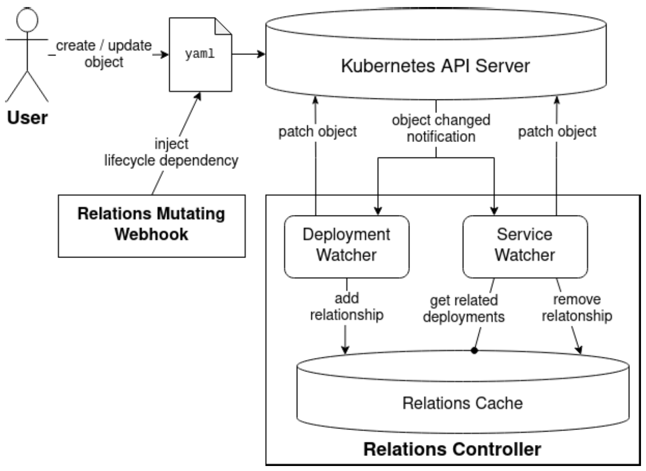


**FIGURE 3.** Architectural overview of orcon. The relations mutating Webhook injects lifecycle dependencies before the objects are persisted in the API server. The relations controller modifies deployments and services in order to establish and update requested relationships.

- The *keyconfig* map describes for each relationship key, how to inject this key into the consuming object using *injectionMethod*. Orcon currently supports injecting relationship values using environment variables and mounted volumes.

Below is an example of a ConsumerConfig mapping three relationship keys to two environment variables and a volume. It also describes that the object should only start after the related object has started.

```
apiVersion: relations.orcon.example/v1alpha1
kind: ConsumerConfig
metadata:
  name: mysql-consumer-config
config:
  lifecycledep: start
  keyConfig:
    url:
      targetKeyName: mysqlurl
      injectionMethod: env
    username:
      targetKeyName: mysqlusername
      injectionMethod: env
    password:
      targetKeyName: pwd
      injectionMethod: volume
      mountPath: /etc/mysql
```

Finally, objects specify which configuration they use for a certain interface with the optional *orcon.dev/config* annotation.

## B. INJECTING RELATION DATA

Figure 3 shows the architecture of the orcon services responsible for taking the appropriate *actions* to establish and manage relationships. The extraction and injection of relation data is managed by the Relations Controller. This service implements the Kubernetes Controller pattern [39], [40] in order to plug into the Kubernetes management plane. Controllers are Docker containers running inside of the Kubernetes cluster that use the Kubernetes API to observe and modify resources.

The relations controller consists of three parts: A Deployment Watcher, a Service Watcher and a Relations Cache. The Relations Cache maps the names of providing objects to objects that request a relationship to them. The sequence diagram in Figure 4 shows that the Deployment Watcher updates the Relations Cache every time a Deployment gets added. The Relations Cache is then used by the Service Watcher to figure out which Deployments to update when a Service changes. The ProviderConfig and ConsumerConfig are used to determine which specific actions to take and how to update related objects. These watchers perform very similar functionality when Deployments and Services change after creation.

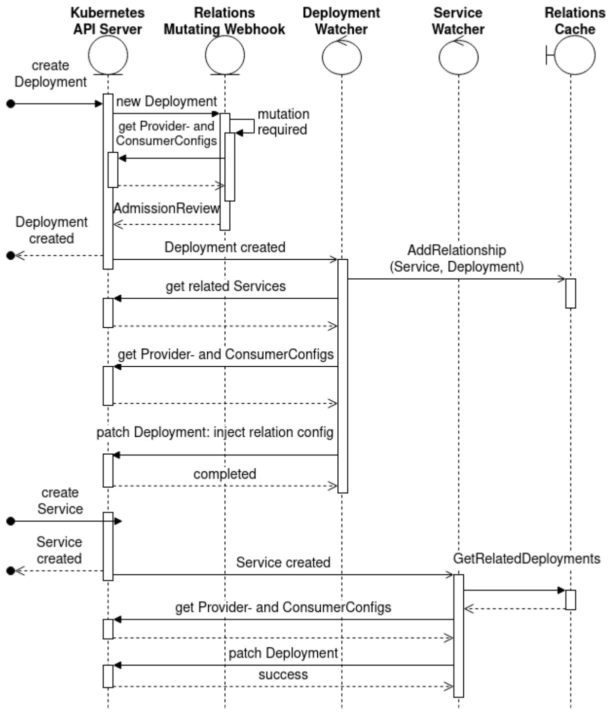


**FIGURE 4.** The relations mutating Webhook injects lifecycle dependencies before the objects are persisted in the API server. The relations controller modifies deployments and services in order to establish and update requested relationships.

### C. INJECTING LIFECYCLE DEPENDENCIES

The orcon Relations Mutating Webhook is responsible for injecting lifecycle dependencies into Deployments as shown in Figure 3. This service implements the Mutating Admission Controller pattern [39], [41] in order to have the ability to change Deployments *before* they are persisted in the API server. Thus, the lifecycle dependencies are injected before the Kubernetes services responsible for deploying Pods can view them. This avoids a race condition where Kubernetes deploys Pods before the lifecycle dependencies are injected.

The lifecycle dependencies themselves take the form of Kubernetes init containers that wait until they receive a signal that the dependent object has started. Since the main container of a Pod will only run after all init containers have exited, adding such an init container effectively halts the execution of the main container until the lifecycle dependency is met. This way, orcon makes sure that all objects start in the correct order.

Figure 4 shows the sequence of the *orcon* admission controller when a new Deployment is created. The controller checks if the Deployment consumes an interface. If so, the controller injects the *orcon* Init Container into the pod template of the Deployment and configures it according to the requirements of the interface. As a result, the application containers of that Deployment will only start after the relation information is added by the relations controller. Important to note here is that the relations controller will only spring into action when a valid relationship is requested and is possible between two objects. The admission controller, however, will inject the lifecycle dependencies immediately, even if the requested relationship is not possible. This ensures that, in the event the providing object is not yet present in the API server, the consuming object will not start. If an object has a lifecycle dependency on another object, it should not start if that other object is not present.

Although it is technically possible for the Admission Controller to inject relationship data, orcon avoids it to reduce latency because Admission Controllers block the acceptance of an object until they are finished. Regular controllers, on the other hand, work concurrently with other operations on those objects thanks to Kubernetes' optimistic concurrency control [7], [42].

### D. OPTIMIZATION

As explained in Section VI-A, orcon heavily uses Kubernetes object annotations to store metadata. Since annotation contents are not indexed by the Kubernetes API, it is not possible to select objects based on them. In order to find all objects which consume a certain relationship, orcon needs to request all objects having any relationship and manually search through the annotations itself.

The first step in reducing the overhead of this process is to cache information of related objects locally in the controller so the Kubernetes API doesn't have to be contacted in order to retrieve information. For this, orcon uses the SharedIndexInformer of the Kubernetes controller SDK. By accessing this eventually-consistent cache directly, orcon avoids expensive calls to the Kubernetes API. Since the Kubernetes API itself is also an eventually-consistent system, orcon natively supports this paradigm without modifications.

The second step in reducing the overhead of object annotations is the RelationsCache. This orcon-specific data structure maps the names of providing objects to cached versions of all objects which consume a relationship with them. This way, when a providing object is updated, finding all objects to whom the change needs to be propagated is an O(1) operation which happens in the controller itself without contacting the Kubernetes API. Every time a relationship changes, orcon updates the RelationsCache to reflect those changes.

The controller itself currently does not support any parallelism. All updates to Kubernetes objects are processed sequentially, one by one.

## VII. EVALUATION

This evaluation compares the performance of the *orcon* orchestrator presented in the previous section to regular Kubernetes and to Juju on Kubernetes, in order to answer RQ 4.

The leftmost part of Figure 5 shows the evaluated conceptual topology. This test case is based on the use-case described in Section III: a number of separate app teams each provide a single application that runs on the Kubernetes cluster managed by the platform team and connects to a single eternal SSE server managed by the core team. The grey and dotted parts of Figure 5 show which components are added with each additional app team.

The *sse* relationship in this topology has the following components:

- *Communication:* The SSE client connects to the SSE server to receive and process events.
- *Lifecycle:* The SSE client can only start *after* the SSE server has started.
- *Configuration:* The SSE client receives the DNS name of the SSE server.

Although orcon supports much more complex relationships and topologies, this evaluated use-case is intentionally simplified for clarity purposes.

### A. SETUP

All performance benchmarks are executed on a vanilla Kubernetes cluster from the Charmed Distribution of Kubernetes (CDK) version 1.14.1 [43] connected to a Ceph cluster for persistent storage. All software is deployed in virtual machines on a VMWare ESXI cluster [44] and managed by Juju 2.5.4 [45]. Each solution is tested with an increasing number of consumers, from 5 to 55 with an increment of 5. Each combination is tested 20 times. The graphs show all measurements as individual dots and crosses. The full source code for the different implementations, the evaluation

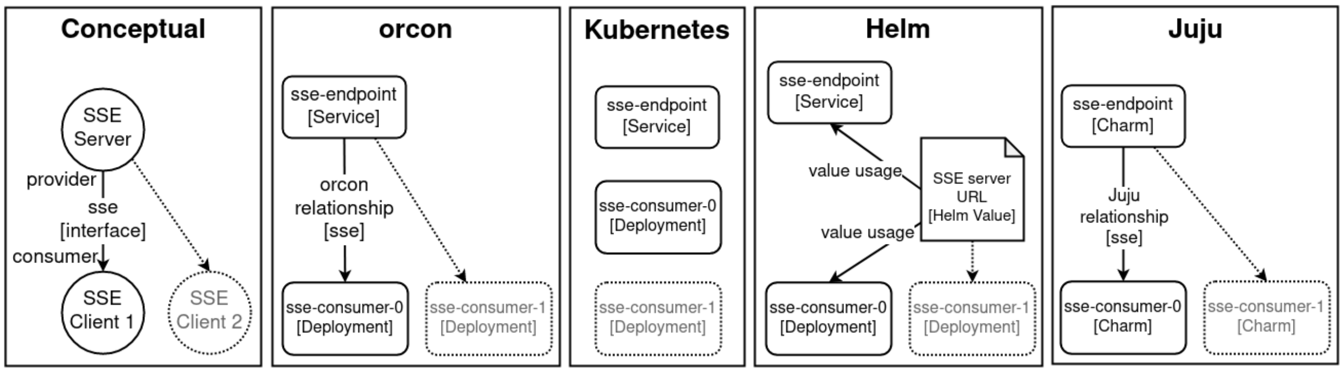


**FIGURE 5.** **Overview of the conceptual topology of the evaluated use-case and the resulting topologies of its implementation using each evaluated solution. The dotted graphics show the components needed to add an additional consumer.**

and the full specification of the test cluster is available on GitHub [46].

### B. EVALUATED SOLUTIONS

These evaluations compare four different solutions to the use-case described in Section III.

1) The "*orcon*" solution deploys the consumers as described in Section VI.
2) The "pure k8s" solution deploys the consumer containers by submitting a deployment.yaml using the kubectl command-line client. The URL is specified using a ConfigMap. The URL is updated by submitting a new deployment.yaml file with the updated URL.
3) The "Helm" solution deploys the same setup as the "pure k8s" solution but the Deployment is templated so that the number of consumers and the URL are specified using Values.
4) The "Juju" solution deploys the consumers using Kubernetes Charms. Each Consumer is a k8s charm that deploys the consumer container and the SSE service is represented by a proxy charm that contains the URL to the SSE service. This URL is transferred to the consumer charms using a Juju relationship. The URL is updated by changing the configuration of the SSE service charm, which then sends the updated URL to all the consumer charms, which in their turn update the PodSpec.

Figure 5 shows the evaluated conceptual topology and the resulting implementation in each solution. The specific models and implementations of each solution are available on GitHub [46].

Note: even though Juju is used to manage the Kubernetes cluster itself, only the "Juju" solution uses Juju for the deployment of the consumers. The other solutions simply deploy on top of the Juju-managed Kubernetes clusters.

### C. FUNCTIONAL EVALUATION

With the goal of comparing the functionality of orcon to the state of the art, we used BPMN 2.0 choreography diagrams to model the interactions required to deploy and update the aforementioned setups. For clarity purposes, interactions where at least one party is a person have a solid border, interactions where both parties are people have an icon in the description, and interactions solely between software systems have a dashed border. We will mainly focus this evaluation on interactions involving humans since those have a significant penalty in terms of latency and potential for mistakes. The performance of the machine-to-machine interactions are benchmarked in Section VII-D.

Figure 6 shows the processes required to create a new app in each solution. These processes assume the appropriate actions have already been taken to add the SSE server to the setup. Helm has the significant downside that it requires the platform team to submit the application on behalf of the app team. Due to Helm's lack of independence in relationships, the entire setup, including the service and existing applications from other teams, needs to be managed as a single entity and only the platform team has the permissions to do this.

Juju has the downside that the app team cannot interact with Kubernetes directly, Juju serves as an intermediary. This shows the advantage of the more integrated approach of orcon: it adds additional abstractions without an additional abstraction layer. Although the Juju solution requires three manual interactions instead of two, this is simply due to Juju's mandatory security concerning relationships that cross management domains. This difference would not exist if all solutions provided the same level of security.

Figure 7 shows the processes required to update an existing app. Here too, Helm suffers from its lack of independence: the app team needs to ask the platform team to update the app on their behalf. Not only does this prevent the app team from interacting with Kubernetes directly, it adds an additional manual step in the process. Juju also has the same downside that its additional abstraction layer prevents the app team from interacting with Kubernetes directly.

Figure 8 shows the role relationships can play in automated response to changes. In orcon and Juju, only the core team has to perform a manual interaction to update the service. The

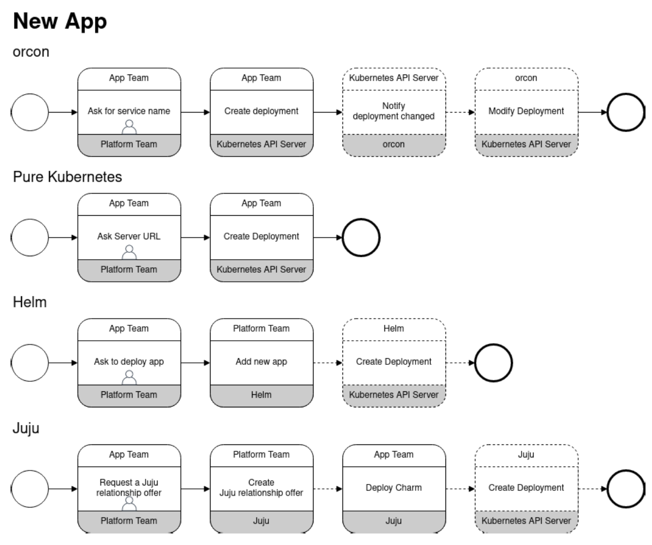


FIGURE 6. Like any TOSCA-based solution, Juju has the downside that users cannot interact with Kubernetes directly, Juju serves as an intermediary.

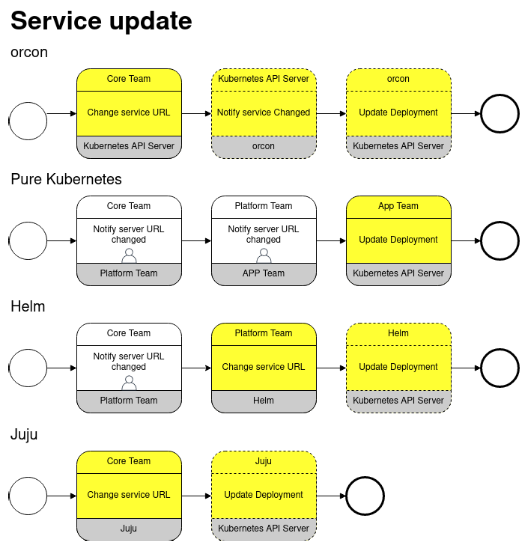


FIGURE 8. With the orcon and Juju setup, the relationship causes the service change to propagate automatically to connected apps without human interaction. The yellow interactions in these diagrams are benchmarked in the performance evaluation.

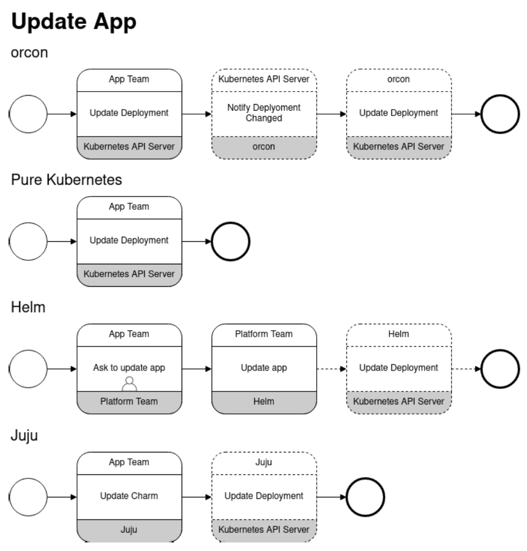


FIGURE 7. The Helm setup requires the platform team to change an App because it manages the entire setup as a single entity.

system then automatically propagates changes to the related Deployments. While Helm dependencies could offer a similar function, it still requires an additional manual action because only the platform team has the permissions required to use it. Juju still has the same downside that the Kubernetes API is hidden from the users. The Pure Kubernetes setup requires the most manual actions due to having no automated way to propagate changes.

### D. PERFORMANCE EVALUATION

With the goal of investigating the overhead of orcon, we benchmarked the time it takes, after a service is updated, to propagate that change to all Deployments. Note, however, that this benchmark does not take into account steps that require human-to-human interaction, so for the "helm" and "pure k8s" solutions not all steps required to update a service are benchmarked. The steps included in this benchmark are highlighted in yellow in Figure 8.

The *orcon* solution proposed in this paper propagates the URL change substantially quicker than Juju. As Figure 9 shows, *orcon* propagates the change to 55 consumers in 48 seconds on average, while Juju requires 146 seconds. The "Juju agents" plot in this graph shows how long it takes for the Juju agents to become ready to process a new change. There is a period of 100 seconds between when Juju updates 55 consumers and when the agents are ready to accept new changes. This is because, for stateless services, the Juju management agent lifecycle is connected to the pod lifecycle. This means that each time a management agent updates a pod, both the pod and the management agent itself shut down and are replaced. As a result, there is a long period after the consumers are updated where the new Juju agent cannot accept new changes because it is initializing. The pure k8s and *orcon* solutions do not have such a cooldown period: these immediately accept new changes after updating the consumers.

Figure 10 dives deeper into the difference between our *orcon* approach, Helm and pure k8s. Since the pure k8s and Helm solutions do not have lifecycle management, the graph also includes a plot of change propagation duration of *orcon* without lifecycle management labelled "orcon without initc". This shows that for 55 consumers, the average additional overhead of lifecycle management using init containers is nine seconds. Although it appears from this graph that the Helm and "pure k8s" solutions are significantly faster than orcon, this does not take into account the manual steps that require human-to-human interactions. These interactions are

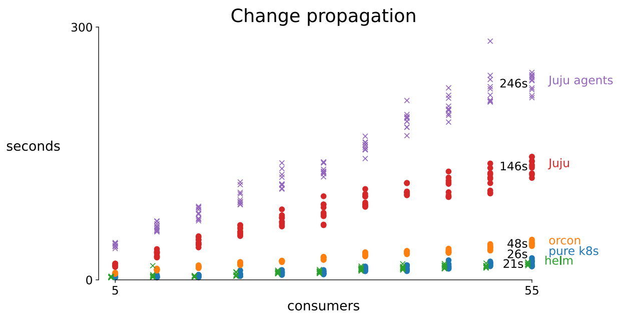


**FIGURE 9.** The *orcon* orchestrator proposed in this paper is significantly faster than Juju in propagating the new URL of the SSE server to the SSE consumers. Moreover, after all consumers are updated, *orcon* is immediately ready to accept new changes while Juju requires a cooldown period during which it cannot propagate URL changes.

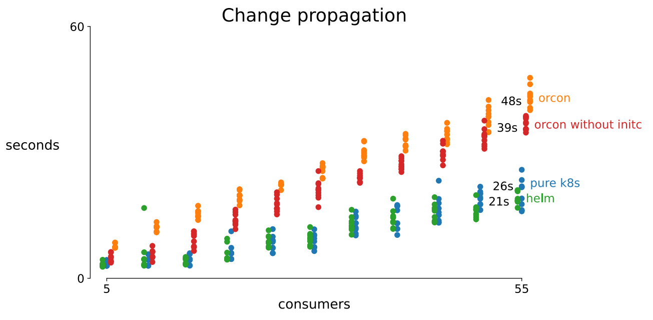


**FIGURE 10.** The init container used by *orcon* for lifecycle management, adds on average an additional 9 seconds to the time for a URL change in the SSE server to be propagated to all consumers.

error-prone and introduce a highly-variable latency that can easily exceed the less than thirty seconds delay between orcon and the state of the art.

Figure 10 also shows that Helm, on average, performs slightly better than the "pure k8s" solution. There are a number of possible explanation for this behavior. The "pure k8s" solution uses a simple method to resubmit the entire application, including all the components that did not change, directly to the Kubernetes API. Helm, on the other hand, has intimate knowledge about what exactly changed due to the use of Helm Values. This might cause helm to interact with the Kubernetes API in a smarter way so as to only change the objects that are actually changed. This behavior was not investigated further because the main focus on this paper is on the performance of orcon compared to the state of the art. Performance between the different state of the art solutions themselves is of less significance to this research.

### E. SUMMARY

Table 2 shows a summarized comparison of orcon with the state of the art. For every process, orcon and Juju have the lowest number of human-to-human interactions required. Although Helm succeeds in reducing the number of human-to-human interactions needed to update a service, it still requires one such interaction because of Helm's monolithic approach to dependencies.

**TABLE 2.** The evaluation shows orcon provides all the benefits of service relationships on Kubernetes while completely integrating into the Kubernetes ecosystem and providing much better performance than Juju.

| Solution | **orcon** | K8s | Helm | Juju |
|---|---|---|---|---|
| **Workflow: number of manual interactions** | | | | |
| New app | 1 | 1 | 1 | 1 |
| Update app | 0 | 0 | 1 | 0 |
| Update Service | 0 | 2 | 1 | 0 |
| **Performance: overhead compared to K8s (s/pod)** | | | | |
| Change propagation | 0.44 | / | 0 | 4.4 |
| **Kubernetes Nativity** | | | | |
| K8s API access | ✓ | ✓ | ✓ | × |
| Part of K8s API | ✓ | ✓ | × | × |
| **Relationship support** | | | | |
| Explicit | ✓ | × | ✓ | ✓ |
| Typed | ✓ | × | × | ✓ |
| Isolated | ✓ | × | × | ✓ |
| Independent | ✓ | × | × | ✓ |
| Communication comp. | ✓ | ✓ | ✓ | ✓ |
| Lifecycle comp. | ✓ | × | × | ✓ |
| Configuration comp. | ✓ | × | ✓ | ✓ |

Orcon has an order of magnitude less overhead compared to Juju. Although orcon appears to have a slight overhead of less than half a second per pod compared to Helm, this is negated by the previously shown fact that helm still requires a human-to-human interaction for this process.

Although Juju makes dependencies explicit and allows the app team to update their service independently of other teams, it adds an additional abstraction layer that makes it impossible to use the full power of the Kubernetes API. Because Juju and Helm objects are not modeled in the Kubernetes API server itself, it is not possible to use other Kubernetes tools to create and manage these objects. Orcon, on the other hand, allows users to directly access the Kubernetes API server and is completely implemented inside of it. The power of this last feature is shown by the fact that other Kubernetes tools, including Helm, can be used to interact with orcon. Orcon is thus not strictly a competitor to Helm, since orcon can be used to enrich a Helm-based approach with true relationship-based dependencies.

## VIII. DISCUSSION

RQ 1 asks "*On an abstract level, what concepts enable modeling and automated management of dependencies between services*"

These concepts are laid out in Section IV, starting with the definition of service relationships: "*An explicit typed connection between isolated and independent service models that enables exchange of configuration information, synchronization of lifecycles and runtime communication*". Furthermore, the concepts "interface" and "role" described in that section are required to model the full extent of both active and possible dependencies between services in a way that both humans and machines can easily understand and reason with them. By implementing these concepts in orcon and evaluating its functionality, we show these concepts indeed make it possible to model a service relationship and take full advantage of its benefits.

RQ 2 asks "*To what extend does the state of the art support modeling and automated management of such dependencies in Kubernetes?*"

Using the concepts of the previous answer, we evaluated the state of the art in Section V and came to the conclusion that, although TOSCA-based solutions offer full support for service relationships on Kubernetes, they fail to allow users access to the underlying orchestrator.

RQ 3 asks "*How can existing platforms be extended in order to support service relationships without hiding the underlying API of the platform to users and without adding extra components in the data path?*"

Section VI implements *orcon* shows how to use the introduced concepts for implementing service relationship support while maintaining user access to the underlying orchestrator. The orchestrator does this by injecting additional abstractions into the Kubernetes API, instead of wrapping it. As a result, orcon users can still take full advantage of the Kubernetes API, and existing Kubernetes ecosystem tools can be used to drive orcon. The *orcon* framework actively resolves dependencies between services and automatically propagates changes in them. The evaluation in Section VII includes a confirmation of this functionality and its advantages for developer workflows.

RQ 4 asks "*What is the orchestration overhead introduced by adding support for such relationships*"

Section VII shows that, although adding these concepts adds a slight orchestration overhead of 0.44 seconds per consumer compared to manual configuration, it removes the need for manual human-to-human interactions, ultimately reducing the total time needed to update services. Moreover, the overhead of orcon is an ten times smaller than that of Juju. This evaluation also show that resolving lifecycle dependencies at the container orchestration level also adds additional overhead. It is thus advised to modify the services to resolve their own lifecycle dependencies at runtime. This has the added benefit that it makes the services more resilient to dependencies breaking after the initial deployment.

Although Juju has much more overhead compared to orcon, it's important to note its much broader feature-set. Juju supports automatic cross-cluster relationships, allows extensive modeling and management of services in and beyond Kubernetes and is network and storage-aware. In cases where orcon's deep integration within the Kubernetes ecosystem is not needed and standardization on a single management tool is possible, Juju can be considered a powerful alternative to add relationship support to Kubernetes. Interesting to note is that, according to our evaluation, about half of the overhead of Juju is caused by a single design decision, namely replacing management agents when the pods they manage restart. This suggests the performance differences between Juju and orcon might not be inherent to Juju's expanded feature-set.

## IX. CONCLUSION

This research proposes orcon, an orchestrator that adds native support of relationships to Kubernetes. It is the first orchestrator that does so without hiding the underlying API and integrating in a way that supports the existing ecosystem of kubernetes tools. Our evaluation shows orcon propagates change at an average of 0.44 seconds per service, an order of magnitude faster than the state of the art.

An interesting future research opportunity is to investigate the overhead difference between orcon and Helm to shed light on possible optimization routes. Another interesting route to explore is to save historical relationship data in order to support easy rollback to previous configurations. This is not possible in the current version of orcon as relationship updates are destructive in the sense that they overwrite previous values. A third opportunity lies in support for Federated Kubernetes clusters. Although the current implementation technically allows creating a relationship to a service in another Kubernetes cluster using the *externalName* functionality explained in Section VI, this still requires manual modification of representative Service objects. An improvement in this area would allow completely automated management of application topologies spanning multiple Kubernetes clusters, opening the door for full topology-based management from the cloud to the edge. Finally, orcon currently only supports relationships between equal peers. Investigating support for hierarchical relationships is an interesting path forward as it could enable the creation of higher-level abstractions inside of the Kubernetes API.

## ACKNOWLEDGMENT

The authors wish to thanks Lennart Onghena for his valuable contributions to orcon.

## REFERENCES

[1] J. Santos, T. Vanhove, M. Sebrechts, T. Dupont, W. Kerckhove, B. Braem, G. V. Seghbroeck, T. Wauters, P. Leroux, S. Latre, B. Volckaert, and F. D. Turck, "City of things: Enabling resource provisioning in smart cities," *IEEE Commun. Mag.*, vol. 56, no. 7, pp. 177–183, Jul. 2018.

[2] Kubernetes Project. (2021). *Kubernetes: Production-Grade Container Orchestration*. [Online]. Available: https://kubernetes.io/

[3] L. Harzenetter, U. Breitenbücher, M. Falkenthal, J. Guth, C. Krieger, and F. Leymann, "Pattern-based deployment models and their automatic execution," in *Proc. IEEE/ACM 11th Int. Conf. Utility Cloud Comput. (UCC)*, Dec. 2018, pp. 41–52.

[4] C. Fehling, F. Leymann, R. Retter, W. Schupeck, and P. Arbitter, *Cloud Computing Patterns: Fundamentals to Design, Build, and Manage Cloud Applications*. Vienna, Austria: Springer-Verlag, 2014. [Online]. Available: https://www.springer.com/gp/book/9783709115671

[5] B. D. Martino, A. Esposito, and G. Cretella, "Semantic representation of cloud patterns and services with automated reasoning to support cloud application portability," *IEEE Trans. Cloud Comput.*, vol. 5, no. 4, pp. 765–779, Oct. 2017.

[6] E. Casalicchio, "Container orchestration: A survey," in *Systems Modeling: Methodologies and Tools* (EAI/Springer Innovations in Communication and Computing), A. Puliafito and K. S. Trivedi, Eds. Cham, Switzerland: Springer, 2019, pp. 221–235, doi: 10.1007/978-3-319-92378-9_14.

[7] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, omega, and Kubernetes: Lessons learned from three container-management systems over a decade," *Queue*, vol. 14, no. 1, pp. 70–93, Jan. 2016, doi: 10.1145/2898442.2898444.

[8] C. A. Ardagna, V. Bellandi, P. Ceravolo, E. Damiani, M. Bezzi, and C. Hebert, "A model-driven methodology for big data analytics-as-a-service," in *Proc. IEEE Int. Congr. Big Data (BigData Congress)*, Jun. 2017, pp. 105–112.

[9] A. Bhattacharjee, Y. Barve, A. Gokhale, and T. Kuroda, "A model-driven approach to automate the deployment and management of cloud services," in *Proc. IEEE/ACM Int. Conf. Utility Cloud Comput. Companion (UCC Companion)*, Dec. 2018, pp. 109–114.

[10] J. Bellendorf and Z. A. Mann, "Cloud topology and orchestration using TOSCA: A systematic literature review," in *Proc. ESOCC*, 2018, pp. 207–215.

[11] A. Palesandro, M. Lacoste, N. Bennani, C. Ghedira-Guegan, and D. Bourge, "Mantus: Putting aspects to work for flexible multi-cloud deployment," in *Proc. IEEE 10th Int. Conf. Cloud Comput. (CLOUD)*, Jun. 2017, pp. 656–663.

[12] K. Project. *Kustomize–Kubernetes Native Configuration Management*. Accessed: Jun. 24, 2021. [Online]. Available: https://kustomize.io/

[13] H. Authors and The Linux Foundation. *Helm*. Accessed: Jun. 24, 2021. [Online]. Available: https://helm.sh/

[14] B. Grant. (Feb. 2018). *[Public] Felloe, Spokes, Axel, and Tiller: A Look at the Parts of Helm*. [Online]. Available: https://docs.google.com/presentation/d/10dp4hKciccincnH6pAFf7t31s82iNvtt_mwhlUbeCDw/edit?usp=embed_facebook

[15] A. Bergmayr, U. Breitenbücher, N. Ferry, A. Rossini, A. Solberg, M. Wimmer, G. Kappel, and F. Leymann, "A systematic review of cloud modeling languages," *ACM Comput. Surv.*, vol. 51, no. 1, p. 22, Feb. 2018, doi: 10.1145/3150227.

[16] L. M. Vaquero, F. Cuadrado, Y. Elkhatib, J. Bernal-Bernabe, S. N. Srirama, and M. F. Zhani, "Research challenges in nextgen service orchestration," *Future Gener. Comput. Syst.*, vol. 90, pp. 20–38, Jan. 2019. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167739X18303157

[17] E. Chirivella-Perez, J. M. A. Calero, Q. Wang, and J. Gutiérrez-Aguado, "Orchestration architecture for automatic deployment of 5G services from bare metal in mobile edge computing infrastructure," *Wireless Commun. Mobile Comput.*, vol. 2018, pp. 1–18, Nov. 2018. [Online]. Available: https://www.hindawi.com/journals/wcmc/2018/5786936/

[18] M. Sebrechts, S. Borny, T. Vanhove, G. Van Seghbroeck, T. Wauters, B. Volckaert, and F. De Turck, "Model-driven deployment and management of workflows on analytics frameworks," in *Proc. IEEE Int. Conf. Big Data (Big Data)*, Dec. 2016, pp. 2819–2826.

[19] M. Hahn, U. Breitenbücher, O. Kopp, and F. Leymann, "Modeling and execution of data-aware choreographies: An overview," *Comput. Sci.-Res. Develop.*, vol. 33, nos. 3–4, pp. 329–340, Aug. 2018, doi: 10.1007/s00450-017-0387-y.

[20] S. Wagner, U. Breitenbücher, O. Kopp, A. Weiß, and F. Leymann, "Fostering the reuse of TOSCA-based applications by merging BPEL management plans," in *Cloud Computing and Services Science*. Cham, Switzerland: Springer, Apr. 2016.

[21] M. Sebrechts, G. Van Seghbroeck, T. Wauters, B. Volckaert, and F. De Turck, "Orchestrator conversation: Distributed management of cloud applications," *Int. J. Netw. Manage.*, vol. 28, no. 6, p. e2036, Nov. 2018. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/nem.2036

[22] M. Zimmermann, U. Breitenbucher, M. Falkenthal, F. Leymann, and K. Saatkamp, "Standards-based function shipping–how to use TOSCA for shipping and executing data analytics software in remote manufacturing environments," in *Proc. IEEE 21st Int. Enterprise Distrib. Object Comput. Conf. (EDOC)*, Oct. 2017, pp. 50–60.

[23] A. Brogi, D. Neri, and J. Soldani, "Freshening the air in microservices: Resolving architectural smells via refactoring," in *Service-Oriented Computing–(ICSOC) Workshops* (Lecture Notes in Computer Science), S. Yangui, A. Bouguettaya, X. Xue, N. Faci, W. Gaaloul, Q. Yu, Z. Zhou, N. Hernandez, and E. Y. Nakagawa, Eds. Cham, Switzerland: Springer, 2020, pp. 17–29.

[24] F. Sivrikaya, N. Ben-Sassi, X. Dang, O. C. Görür, and C. Kuster, "Internet of smart city objects: A distributed framework for service discovery and composition," *IEEE Access*, vol. 7, pp. 14434–14454, 2019.

[25] B. Hirsch, T. Konnerth, and A. Heßler, "Merging agents and services–the JIAC agent platform," in *Multi-Agent Programming: Languages, Tools and Applications*, A. El Fallah Seghrouchni, J. Dix, M. Dastani, and R. H. Bordini, Eds. Boston, MA, USA: Springer, 2009, pp. 159–185, doi: 10.1007/978-0-387-89299-3_5.

[26] A. Javed, S. Kubler, A. Malhi, A. Nurminen, J. Robert, and K. Framling, "BIoTope: Building an IoT open innovation ecosystem for smart cities," *IEEE Access*, vol. 8, pp. 224318–224342, 2020.

[27] P. Štefanič, M. Cigale, A. C. Jones, L. Knight, I. Taylor, C. Istrate, G. Suciu, A. Ulisses, V. Stankovski, S. Taherizadeh, G. F. Salado, S. Koulouzis, P. Martin, and Z. Zhao, "SWITCH workbench: A novel approach for the development and deployment of time-critical microservice-based cloud-native applications," *Future Gener. Comput. Syst.*, vol. 99, pp. 197–212, Oct. 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X1831094X

[28] V. Bracke, M. Sebrechts, B. Moons, J. Hoebeke, F. De Turck, and B. Volckaert, "Design and evaluation of a scalable Internet of Things backend for smart ports," *Softw., Pract. Exper.*, pp. 1557–1579, Apr. 2021. [Online]. Available: https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.2973

[29] IDLab (Ghent University, IMEC). *Obelisk*. Accessed: Jun. 24, 2021. [Online]. Available: https://obelisk.ilabt.imec.be/api/v2/docs/

[30] G. Muntoni, J. Soldani, and A. Brogi, "Mining the architecture of microservice-based applications from their Kubernetes deployment," in *Advances in Service-Oriented and Cloud Computing* (Communications in Computer and Information Science), C. Zirpins, I. Paraskakis, V. Andrikopoulos, N. Kratzke, C. Pahl, N. El Ioini, A. S. Andreou, G. Feuerlicht, W. Lamersdorf, G. Ortiz, W.-J. Van den Heuvel, J. Soldani, M. Villari, G. Casale, and P. Plebani, Eds. Cham, Switzerland: Springer, 2021, pp. 103–115.

[31] Istio Authors. *Istio*. Accessed: Jun. 24, 2021. [Online]. Available: https://istio.io/latest/

[32] Dynatrace LLC. (2021). *Dynatrace | The Leader in Automatic and Intelligent Observability*. [Online]. Available: https://www.dynatrace.com/

[33] W. Chareonsuk and W. Vatanawood, "Translating TOSCA model to Kubernetes objects," in *Proc. 18th Int. Conf. Electr. Eng./Electron., Comput., Telecommun. Inf. Technol. (ECTI-CON)*, May 2021, pp. 311–314.

[34] M. Bogo, J. Soldani, D. Neri, and A. Brogi, "Component-aware orchestration of cloud-based enterprise applications, from TOSCA to Docker and Kubernetes," *Softw., Pract. Exper.*, vol. 50, no. 9, pp. 1793–1821, 2020. [Online]. Available: https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.2848

[35] A. Borisova, V. Shvetcova, and O. Borisenko, "Adaptation of the TOSCA standard model for the Kubernetes container environment," in *Proc. Ivannikov Memorial Workshop (IVMEM)*, Sep. 2020, pp. 9–14.

[36] M. Rutkowski, C. Lauwers, C. Noshpitz, and C. Curescu, "TOSCA simple profile in YAML version 1.3," OASIS Open, Woburn, MA, USA, OASIS Standard, Feb. 2020. [Online]. Available: https://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.3/os/TOSCA-Simple-Profile-YAML-v1.3-os.html and https://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.3/TOSCA-Simple-Profile-YAML-v1.3.html

[37] M. Sebrechts, C. Johns, G. Van Seghbroeck, T. Wauters, B. Volckaert, and F. De Turck, "Beyond generic lifecycles: Reusable modeling of custom-fit management workflows for cloud applications," in *Proc. IEEE 11th Int. Conf. Cloud Comput. (CLOUD)*, Jul. 2018, pp. 326–333.

[38] M. Sebrechts, S. Borny, and L. Onghena. (Feb. 2021). *IBCNServices/Orcon*. [Online]. Available: https://github.com/IBCNServices/orcon

[39] M. Hausenblas and S. Schimanski, *Programming Kubernetes: Developing Cloud-Native Applications*, 1st ed. Sebastopol, CA, USA: O'Reilly Media, Aug. 2019.

[40] Kubernetes Project. (Feb. 2021). *Kubernetes Documentation: Controllers*. [Online]. Available: https://kubernetes.io/docs/concepts/architecture/controller/

[41] Kubernetes Project. (Feb. 2021). *Kubernetes Documentation: Using Admission Controllers*. [Online]. Available: https://kubernetes.io/docs/reference/access-authn-authz/admission-controllers/

[42] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes, "Omega: Flexible, scalable schedulers for large compute clusters," in *Proc. 8th ACM Eur. Conf. Comput. Syst.*, New York, NY, USA, Apr. 2013, pp. 351–364, doi: 10.1145/2465351.2465386.

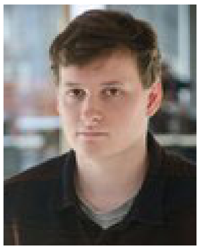
[43] Canonical Ltd. *Charmed Kubernetes | Juju*. Accessed: Jun. 24, 2021. [Online]. Available: https://jaas.ai/canonical-kubernetes

[44] VMWare. *What is ESXI? | Bare Metal Hypervisor | ESX*. Accessed: Jun. 24, 2021. [Online]. Available: https://www.vmware.com/products/esxi-and-esx.html

[45] C. Ltd. *JAAS–Juju as a Service | Juju*. Accessed: Jun. 24, 2021. [Online]. Available: https://jaas.ai/

[46] Merlijn Sebrechts. *IBCNServices/Kubernetes-Relationships-Results*. [Online]. Available: https://github.com/IBCNServices/kubernetes-relationships-results

**MERLIJN SEBRECHTS** received the M.Sc. degree in information engineering technology from Ghent University, in July 2015, where he is currently pursuing the Ph.D. degree with imec, IDLab, Department of Information Technology (INTEC). His research interests include simplifying the management of distributed data platforms using service orchestration, agent-based automation, and cloud modeling languages.



**SANDER BORNY** received the M.Sc. degree in information engineering technology from Ghent University, in 2016. He is currently a Research Assistant with IDLab, Ghent University - imec. His focus is providing cloud infrastructure, DevOps and DataOps to various projects in the fields of smart cities, smart grids, and the Internet of Things.

**TIM WAUTERS** (Member, IEEE) received the M.Sc. and Ph.D. degrees in electro-technical engineering from Ghent University, in 2001 and 2007, respectively. He has been working as a Postdoctoral Fellow of F.W.O.-V. with the Department of Information Technology (INTEC), Ghent University. He is currently active as a Senior Researcher at imec. His work has been published in more than 120 scientific publications. His research interests include design and management of networked services, covering multimedia distribution, cybersecurity, big data, and smart cities.

**BRUNO VOLCKAERT** (Member, IEEE) received the Ph.D. degree in resource management for grid computing from Ghent University, in 2006. He is currently a Professor in advanced distributed systems with Ghent University and a Senior Researcher with imec. He has worked on over 45 national and international research projects. He is the author or coauthor of more than 150 peer-reviewed papers published in international journals and conference proceedings. His current research interests include reliable and high performance distributed software systems for A.O. smart cities, scalable cybersecurity detection and mitigation architectures, and autonomous optimization of cloud-based applications.

**FILIP DE TURCK** (Fellow, IEEE) leads the Network and Service Management Research Group, Ghent University, Belgium, and imec. He has coauthored over 700 peer-reviewed papers. His research interests include design of secure and efficient softwarized networks and cloud systems. He was recently elevated as an IEEE Fellow for outstanding technical contributions. He is involved in several research projects with industry and academia, serves as the Chair for the IEEE Technical Committee on Network Operations and Management (CNOM), and a Steering Committee Member of the IFIP/IEEE IM, IEEE/IFIP NOMS, IEEE/IFIP CNSM, and IEEE NetSoft Conferences. He serves as the Editor-in-Chief for IEEE TRANSACTIONS ON NETWORK AND SERVICE MANAGEMENT (TNSM).