Guiding Architectural Decision Making on Service Mesh Based Microservice Architectures

Amine El Malki and Uwe Zdun

University of Vienna, Faculty of Computer Science, Research Group Software Architecture, Austria, Email: {amine.elmalki,uwe.zdun}@univie.ac.at

Abstract. Microservices are becoming the de-facto standard way for software development in the cloud and in service-oriented computing. Service meshes have been introduced as a dedicated infrastructure for managing a network of containerized microservices, in order to cope with the complexity, manageability, and interoperability challenges in especially large-scale microservice architectures. Unfortunately so far no dedicated architecture guidance for designing microservices and choosing among technology options in a service mesh exist. As a result, there is a substantial uncertainty in designing and using microservices in a service mesh environment today. To alleviate this problem, we have performed a model-based qualitative in-depth study of existing practices in this field in which we have systematically and in-depth studied 40 reports of established practices from practitioners. In our study we modeled our findings in a rigorously specified reusable architectural decision model, in which we identified 14 architectural design decisions with 47 decision outcomes and 77 decision drivers in total. We estimated the uncertainty in the resulting design space with and without use of our model, and found that a substantial uncertainty reduction can be potentially achieved by applying our model.

Keywords: Microservices; Service Meshes; Software Design; Software Architecture; Modeling

1 Introduction

Microservices are a recent approach for designing service architectures that evolved from established practices in service-oriented architectures [13,17,28]. As microservices, especially in large-scale systems, introduce many challenges and high complexity in terms of manageability and interoperability, *service meshes* [15] have been introduced as an infrastructure for managing the communication of containerized microservices and perform many related tasks. For this, they usually use a network of lightweight proxies or sidecars that handle all the communication burden [12,16]. As a result, the coupling between microservices and of microservices to the infrastructure services can get drastically reduced. This also eases establishing interoperability between microservices developed in different programming languages and with different technologies. The proxies or sidecars form a *data plane* that is typically managed by a *control plane* [22].

Unfortunately so far no dedicated architectural guidance exists on how to design and architect microservices in a service mesh environment apart from practitioner blogs, industry white papers, experience reports, system documentations, and similar informal literature (sometimes called gray literature). This includes that so far there is no guidance for users of service mesh technologies or even their implementors to select the right design and technology options based on their respective properties. Very often it is even difficult to understand what all the possible design options, their possible combination, and their impacts on relevant quality properties and other decision drivers are. As a result, there is substantial uncertainty in architecting microservices in a service mesh environment, which can only be addressed by gaining extensive personal experience or gathering the architectural knowledge from the diverse, often incomplete, and often inconsistent existing practitioner-oriented knowledge sources.

To alleviate these problems, we have performed a qualitative, in-depth study of 40 knowledge sources in which practitioners describe established practices. We have based our study on the model-based qualitative research method described in [26], which uses such documented practitioner sources as rather unbiased knowledge sources and systematically codes them using established coding and constant comparison methods [6] combined with precise software modeling, in order to develop a rigorously specified software model of established practices and their relations. This paper aims to study the following research questions:

- **RQ1** What are the established practices that commonly appear in service mesh based designs and architectures?
- **RQ2** What are the dependencies of those established practices? Especially which architectural design decisions (ADDs) need to be made in service mesh based designs and architectures?
- **RQ3** What are the decision drivers in those ADDs to adopt the practices?

In addition to studying and answering these research questions, we have estimated the decision making uncertainty in the resulting ADD design space, calculated the uncertainty left after applying the guidance of our ADD model, and compared the two. Our model shows a potential to substantially reduce the uncertainty not only by documenting established practices, but also by organizing the knowledge in a model.

The remainder of this paper is organized as follows: In Section 2 we compare to the related work. Section 3 explains the research method we have applied in our study. Then Section 4 explains a precise specification of the service mesh design decisions resulting from our study. The uncertainty estimation is discussed in Section 5, followed by a discussion in Section 6 and conclusions in Section 7.

2 Related Work

Service meshes have been identified in the literature as the latest wave of service technology [12]. Some research studies use service meshes in their solutions. For example, Truong et al. [23] use a service mesh architecture to reduce rerouting effort in cloud-IoT scenarios. Studies on generic architecture knowledge specific to service meshes are rather rare in the scientific literature so far. One example that considers them is TeaStore, which intrafficControlDecision-troduces a microservice-based reference architecture for cloud researchers and considers practices used in service meshes [4]. More sources can be found on general microservice best practices. For instance, Richardson [20] provides a collection of microservice design patterns. Another set of patterns on microservices has been published by Gupta [8]. Microservice best practices are discussed in [13], and similar approaches are summarized in a recent mapping study [18]. So far, none of these approaches has put specific focus on the service mesh practices documented in our study.

A field of study related to service mesh architectures are studies on microservice decomposition, as this can lead to decision options and criteria related to the topology of the service mesh. While the microservice decomposition itself is studied in the scientific literature extensively (see e.g. [10,1,25]), its influence on the design of the deployment in a service mesh and its topology are studied only rarely. For instance, Zheng et al. [27] study the SLA-aware deployment of microservices. Selimi et al. [21] study the service placement in a microservice architecture. Both studies are not specific for service meshes, but could be applied to them. In contrast to our study which aims to cover a broad variety of architecting problems, these studies only cover a very specific design issue in a microservice architecture.

The model developed in our study can be classified as a reusable ADD model [29]. Decision documentation models have been used by many authors before, and quite a number of them are focused on services, such as those on service-oriented solutions [29], service-based platform integration [14], REST vs. SOAP [19], microservice API quality [26], big data repositories [7], and service discovery and fault tolerance [9]; however, none of them considers service meshes yet.

3 Research Method

This paper aims to systematically study the established practices in the field of service mesh based architectures. A number of methods have been suggested to informally study established practices, including pattern mining (see e.g. [3]). As in our work, we rather aim to provide a rigorously specified model of the established practices, e.g., to support tool building or the definition of metrics and constraints in our future work, we decided to follow the model based qualitative research method described in [26]. It aims to systematically study the established practices in a particular field and is based on the established qualitative research method Grounded Theory (GT) [6] but in contrast to GT it produces inputs for formal software modeling like model element or relation instances, not just informal textual codes. Like GT, we studied each knowledge source in depth. The method uses descriptions of established practices from the so-called gray literature (i.e., practitioner reports, system documentations, practitioner blogs, etc.). These sources are then used as unbiased descriptions of established practices in the further analysis (in contrast to sources like interviews as used in classic GT). We followed a similar coding process, as well as a constant comparison procedure to derive our model as used in GT. In contrast to classical GT, our research began with initial research questions, as in Charmaz's constructivist GT [2]. Whereas GT typically uses textual analysis, we used textual codes only initially and then transferred them into formal UML models.

A crucial question in GT is when to stop this process; here, theoretical saturation [6] has attained widespread acceptance in qualitative research: We stopped our analysis when 5 to 7 additional knowledge sources did not add anything new to our understanding of the research topic. As a result of this very conservative operationalization of theoretical saturation, we studied a rather large number of knowledge sources in depth (40 in total, summarized in Table 1), whereas most qualitative research often saturates with a much lower number of knowledge sources. Our search for knowledge sources was based on popular search engines (e.g., Google, Bing), social network platforms used by practitioners (e.g., Twitter, Medium), and technology portals like InfoQ and DZone.

Proof-of-Concept Implementation Our proof-of-concept implementation is based on our existing modeling tool implementation CodeableModels[1], a Python implementation for precisely specifying meta-models, models, and model instances in code with an intuitive and lightweight interface. We implemented all models described in this paper together with automated constraint checkers and PlantUML code generators to generate graphical visualizations of all meta-models and models.

Table 1: Knowledge Sources Included in the Study

Code	Description	Reference
S1	Istio Prelim 1.2 / Traffic Management (documentation)	`http://bit.ly/2Js3JXj`
S2	Using Istio to support Service Mesh on Multiple . . . (blog)	`http://bit.ly/2FqMce5`
S3	Service mesh data plane vs. control plane (blog)	`http://bit.ly/2EtC8z6`
S4	The Importance of Control Planes with Service Meshes . . . (blog)	`http://bit.ly/2He7JYu`
S5	Envoy Proxy for Istio Service Mesh (documentation)	`https://bit.ly/2HaNdrE`
S6	Our Move to Envoy (blog)	`https://bit.ly/2Vyyefd`
S7	Envoy Proxy 101: What it is, and why it matters? (blog)	`https://bit.ly/2HaNhYq`
S8	Service Mesh with Envoy 101 (blog)	`https://bit.ly/2UjPuVn`
S9	Microservices Patterns With Envoy Sidecar Proxy (blog)	`https://bit.ly/2tOWo9C`
S10	Ambassador API Gateway as a Control Plane for Envoy (blog)	`http://bit.ly/2TuaZWj`
S11	Streams and Service Mesh - v1.0.x | Kong . . . (documentation)	`http://bit.ly/2UGX7W1`
S12	Istio Prelim 1.2 / Security (documentation)	`http://bit.ly/2HyOIkH`
S13	Consul Architecture (documentation)	`https://bit.ly/2ITnhU2`
S14	Global rate limiting — envoy . . . (documentation)	`http://bit.ly/2Js3JXj`
S15	Cilium 1.4: Multi-Cluster Service Routing, . . . (blog)	`http://bit.ly/2Cv49pU`
S16	Proxy Based Service Mesh (blog)	`https://bit.ly/2VzpbL2`
S17	Smart Networking with Consul and Service Meshes (blog)	`http://bit.ly/2Uk14jg`
S18	A sidecar for your service mesh (blog)	`http://bit.ly/2ThMrvF`

[1] `https://github.com/uzdun/CodeableModels`

S19	Istio Prelim 1.2 / Multicluster Deployments (documentation)	`http://bit.ly/2udsxI3`
S20	Microservices Reference Architecture from NGINX (blog)	`http://bit.ly/2U3tNw1`
S21	Comparing Service Mesh Architectures (blog)	`http://bit.ly/2tQ2GWd`
S22	Istio Multicluster on OpenShift – Red Hat OpenShift . . . (blog)	`https://red.ht/2FcMyn4`
S23	Amazon ElastiCache for Redis FAQs (documentation)	`https://amzn.to/2TgGML8`
S24	Service Mesh for Microservices (blog)	`http://bit.ly/2TCd6Is`
S25	Designing microservices: . . . (documentation)	`http://bit.ly/2tPQkO4`
S26	HashiCorp Consul 1.2: Service Mesh (blog)	`http://bit.ly/2Fnj1It`
S27	Connect-Native App Integration (documentation)	`http://bit.ly/2NEDMlL`
S28	Service discovery — envoy . . . (documentation)	`http://bit.ly/2Tfp59H`
S29	Linkerd2 Proxy (open source implementation)	`https://bit.ly/2HaiFqa`
S30	Multi Cluster Support for Service Mesh . . . (blog)	`http://bit.ly/2Jp6isS`
S31	Linkerd Architecture (documentation)	`http://bit.ly/2Uki3lt`
S32	Federated Service Mesh on VMware PKS . . . (blog)	`http://bit.ly/2TNRitD`
S33	Consul vs. Istio (documentation)	`http://bit.ly/2Tdx5gd`
S34	Guidance for Building a Control Plane to Manage Envoy . . . (blog)	`http://bit.ly/2CCAYRU`
S35	Comparing Service Meshes: Linkerd vs. Istio . . . (blog)	`http://bit.ly/2TWQAtT`
S36	Connect - Proxies - Consul by HashiCorp (documentation)	`http://bit.ly/2UViLWG`
S37	Approaches to Securing Decentralised Microservices . . . (blog)	`http://bit.ly/2Wp5Ojn`
S38	Istio Routing Basics – Google Cloud Platform . . . (blog)	`http://bit.ly/2OoRODn`
S39	Integrating Istio 1.1 mTLS and Gloo Proxy . . . (blog)	`http://bit.ly/2UTpctm`
S40	Kubernetes-based Microservice Observability . . . (blog)	`http://bit.ly/2FvE4aT`

4 Service Mesh Design Decisions

Following our study results, we identified 14 ADDs for service meshes described in detail below. *Service Meshes* are usually used together with a *Container Orchestrator* such as Kubernetes or Docker Swarm. That is, the services in the mesh, the central services of the service mesh, and service mesh proxies are usually containerized and the containers are orchestrated. Very often service meshes are used to deal with heterogeneous technology stacks. That is, a major goal is that microservices can be written as HTTP servers with any programming language or technology, and without modification these services get containerized and managed in a mesh, including high-level services like service discovery, load balancing, circuit breaking, and so on. In the first four sections, we describe ADDs that characterize a service mesh as a whole. The remaining section describes ADDs that can be made for specific components of a service mesh.

4.1 Managed Cross-Service Communication Decision

As stated previously, a *Service Mesh* is composed of a set of networked proxies or *Sidecars* that handle the communication between microservices [12,22]. The decision regarding managed communication across the services in a *Service Mesh* is made for the *Service Endpoints* of these microservices, as illustrated in Figure 1. Not using managed cross-service communication is a decision option for each service endpoint but please note that this essentially means to not follow a service mesh architecture for the endpoint. Alternatively, we can select between the two following design options: *Service Proxy* and *API-Based Service Integration*. *Service Proxy* is the commonly supported option. If the *Service Proxy* is hosted in a container that runs alongside the service container (i.e., in the same pod of the *Container Orchestrator*), the service proxy is called a *Sidecar Proxy*. A few service meshes offer the additional option *API-Based Service Integration*, which means that the service uses a service mesh API to register itself in the mesh and is then integrated without a dedicated proxy. The entire cross-service communication handled by proxies or otherwise integrated services is called the *Data Plane* of the *Service Mesh*. Centralized services of the service mesh are usually called the *Control Plane* (discussed below).

The *Service Proxy* option has the benefit to make it easier to protect the service from malicious or overloaded traffic by achieving *access control*, *TLS termination*, *rate limiting and quotas*, *circuit breaking*, *load balancing*, and other tasks; this is discussed in more depth in Section 4.5. Also, the independence of the service from its proxy increases the *extensibility* and *maintainability* of the service, which is, as a result, not aware of the network at large and only knows about its local proxy. However, this option might produce additional *communication overheads* and *congestions*. The major benefit of choosing an *API-Based Service Integration* over a *Service Proxy* is that it makes the service mesh less *complex* and there is less *communication overhead*. However, doing so limits its *extensibility* and *interoperability*. The option not to manage cross-service communication basically means that all benefits of service mesh are not achievable.

An example realizing the *API-Based Service Integration* is Connect Proxy used in Consul that is implemented using language-specific libraries that are used directly in microservices code. *Service Proxy* and *Sidecar Proxies* are more frequently supported; examples are Envoy Proxy [5] in the Istio Service Mesh [11], Kong Proxy, NGINX Proxy and Linkerd Proxy. Most such service proxy technologies can be deployed as a sidecar or a service proxy running in a different environment (e.g., different server or VM); they usually also offer the option to be used as a *Front Proxy* as discussed in the next section.

4.2 Managed Ingress Communication Decision

In addition to handling cross-service communication, service meshes often intercept incoming traffic, usually called ingress traffic. The decision for managed ingress communication is usually made for the *Service Mesh* as a whole. The ingress traffic then needs to be routed to the containers orchestrated in the mesh. Of course, we might choose not to manage ingress communication but this is a risky and dangerous option since it might expose the service mesh to malicious or overloaded traffic. This option may be adopted in case of a private service mesh, but such meshes seem to be very rare. The typical design option chosen is a *Front Proxy* which is used by the *Control Plane* to intercept ingress traffic as shown in Figure 2. An *API Gateway* [20], a common microservice pattern with the goal to provide a common API for a number of services, can be

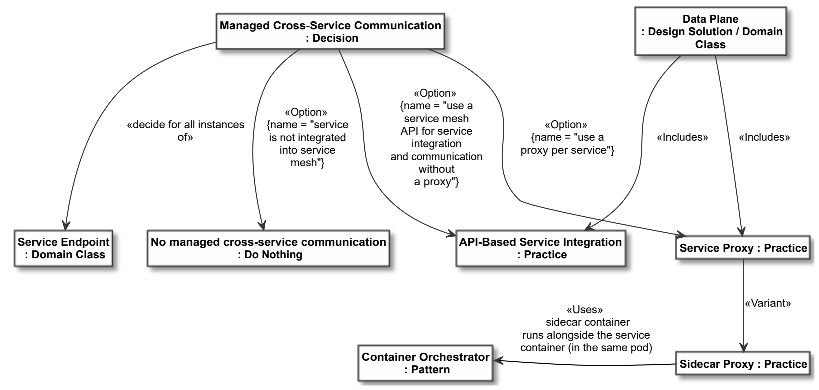
Managed Cross-Service Communication : Decision
Data Plane : Design Solution / Domain Class
«decide for all instances of»
«Option» {name = "service is not integrated into service mesh"}
«Option» {name = "use a service mesh API for service integration and communication without a proxy"}
«Option» {name = "use a proxy per service"}
«Includes»
«Includes»
Service Endpoint : Domain Class
No managed cross-service communication : Do Nothing
API-Based Service Integration : Practice
Service Proxy : Practice
«Variant»
«Uses» sidecar container runs alongside the service container (in the same pod)
Container Orchestrator : Pattern
Sidecar Proxy : Practice

Fig. 1. Managed Cross Service Communication Decision

realized based on a *Front Proxy* of a service mesh. A *Front Proxy* can protect the service mesh from *malicious traffic*. It can provide *proxy tasks* such as *load balancing* and *multi-protocol support* at the perimeter of the service mesh. Clients are *shielded from details about the inner workings* of the service mesh and are provided with an *API at the client-needed granularity*; this reduces *complexity for clients*. The additional proxy increases *complexity for developers* of the service mesh. The *performance of requests* can be increased, as less roundtrips from clients to services are needed, if the *Front Proxy* can retrieve data from multiple services for one request from a client. However, the additional network hop for accessing the *Front Proxy* decreases the *performance*. An example of this type of proxy is the NGINX Ingress Controller. Most of the proxies from the previous section can also be used as *Front Proxies*.

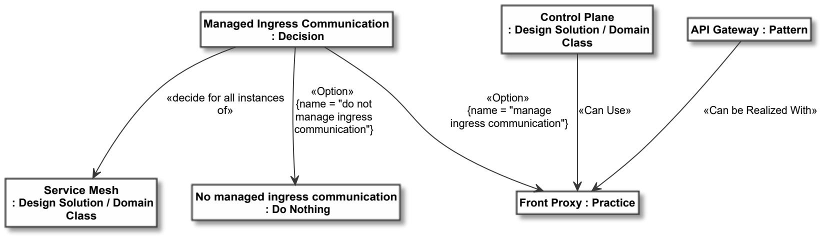
Managed Ingress Communication : Decision
Control Plane : Design Solution / Domain Class
API Gateway : Pattern
«decide for all instances of»
«Option» {name = "do not manage ingress communication"}
«Option» {name = "manage ingress communication"}
«Can Use»
«Can be Realized With»
Service Mesh : Design Solution / Domain Class
No managed ingress communication : Do Nothing
Front Proxy : Practice

Fig. 2. Managed Ingress Communication Decision

4.3 Traffic Control Decision

Communication in service meshes generates a lot of traffic and data that needs to be controlled and captured e.g. to distribute access control and usage policies, and observe and collect telemetry, traces and metrics. The traffic control decision is usually made for the whole *Service Mesh* as illustrated in Figure 3. There are four traffic control options:

– *Centralized Control Plane* – A central component, called the *Control Plane*, controls traffic of a service mesh. It is responsible of managing and configuring sidecars in addition to distributing access control and usage policies, observing and collecting telemetry, traces and metrics, in addition to numerous other services like service discovery, as described in Section 4.5.
– *Distributed Control Plane* – Each service of a service mesh has its own cache that is efficiently updated from the rest of the services. This helps to enforce policies and collect telemetry at the edge.
– *Front Proxy* – The proxy is responsible for intercepting incoming traffic from outside the service mesh as described in Section 4.2. It might also be extended to handle traffic control at the entry point of the service mesh. This option can potentially be combined with the two previous options (for that reasons, the decision is marked with a stereotype that indicates that multiple answers can be selected).
– Finally no dedicated traffic control can be used as well.

The most obvious benefit of using a *Centralized Control Plane* is its *simplicity* and ease of *administration*. However, especially when using one single control plane, it produces a *single point of failure* and is hard to *scale*. Also, it might cause *traffic congestion* which *increases latency*. *Centralized Control Planes* provide policies to the *Service Proxy* on how to perform routing, load balancing and access control. In that case, the next optional decision to take is related to service mesh expansion. Istio service mesh, for example, which is based on *Centralized Control Plane* supports service mesh expansion in a multi-cluster environment. On the other hand, a *Distributed Control Plane* is *highly scalable* and there is *no single point* of failure. However, this option is the most *complex* and thus *risky* option. Using this option, traffic may be forwarded to either a *Service Proxy* or directly to a service via *API-Based Service Integration* as described in Section 4.1. Consul for example, which implements *Distributed Control Plane*, consists of a set of client agents that are efficiently updated from server agents to control traffic at the edge. An example using *Front Proxy*, described in Source S10 in Table 1, uses Envoy which can be extended to become an *API Gateway*, which then can do traffic control for the service mesh by integrating with Istio. The *Front Proxy* solution does not have the *fine-grained control* offered by the other options, as it is only applied in a central place. It is also a *single point of failure* and is negative for *congestion* and *latency*, but is a *simple* and *non-complex* solution. If used together with one of the other options, it increases the *complexity* of these options even further, but enables more *fine-grained control* for the ingress traffic and thus can reduce the overall traffic in the mesh. These options lead us to the next optional decision regarding distributing traffic control and other tasks among *Control Plane* and *Data Plane* (see Section 4.5 for a list of these follow-on decisions, not shown in Figure 3 for brevity). Figure 3 shows these decision options and their relations.

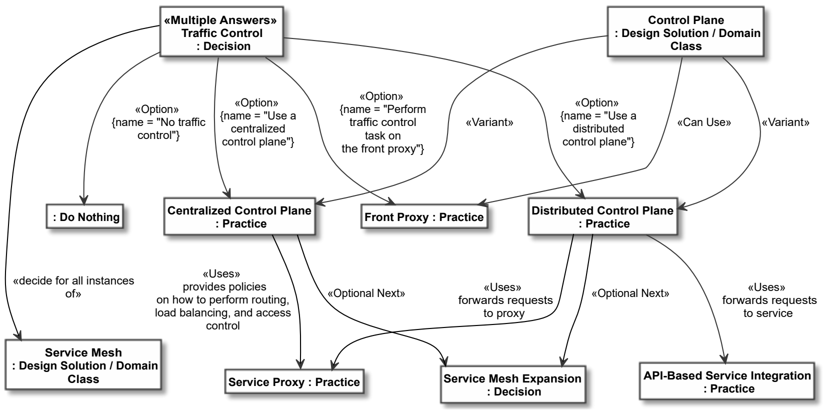
«Multiple Answers» Traffic Control : Decision
Control Plane : Design Solution / Domain Class
«Option» {name = "No traffic control"}
«Option» {name = "Use a centralized control plane"}
«Option» {name = "Perform traffic control task on the front proxy"}
«Variant»
«Option» {name = "Use a distributed control plane"}
«Can Use»
«Variant»
: Do Nothing
Centralized Control Plane : Practice
Front Proxy : Practice
Distributed Control Plane : Practice
«decide for all instances of»
«Uses» provides policies on how to perform routing, load balancing, and access control
«Optional Next»
«Uses» forwards requests to proxy
«Optional Next»
«Uses» forwards requests to service
Service Mesh : Design Solution / Domain Class
Service Proxy : Practice
Service Mesh Expansion : Decision
API-Based Service Integration : Practice

Fig. 3. Traffic Control Decision

4.4 Service Mesh Expansion Decision

To *scale* and achieve *redundancy*, service meshes can be expanded and form multi-clustered service meshes, leading to the selection of the option *Multi-Cluster Support* in the decision illustrated in Figure 4. The service mesh expansion decision is made for the *Service Mesh* itself. Selecting *Multi-Cluster Support* may result in higher *complexity* and increased *network bandwidth* need and *cost*. The decision option *Multi-Cluster Support* is in its simple form just using one *Centralized Control Plane* (see Section 4.3) that controls multiple service meshes. The most obvious benefit of this option is its *simplicity* and *ease of administration*. However, it is creating a *single point of failure* and might produce traffic bottlenecks which increase *latency*. *Multi-Cluster Support* has one variant *Multi-Cluster Support with Multiple Control Planes* with no *single point of failure*. This option variant uses *Multiple Control Planes* which is a variant of *Control Plane* as shown in Figure 4.

Istio Multicluster on Openshift, described in source S22 of Table 1, is an example that implements *Multi-Cluster Support* using one *Centralized Control Plane*. In this example, one cluster is hosting the *Control Plane* and the others host the *Data Plane* as well as some parts of the *Control Plane* for distributing certificates and *Sidecar Proxy* injection. Another example is NSX service mesh, described in source S32 of Table 1, which is also based on Istio but implements the variant *Multi-Cluster Support with Multiple Control Planes* by enabling a local service mesh per Kubernetes cluster.

4.5 Central Services and Proxy Tasks

As explained above, the *Control Plane* and *Data Plane* provide numerous central services and proxy tasks, and many of those are achieved jointly. The decisions on central services and proxy tasks are usually made for the *Service Mesh* itself but can in many cases be changed for individual services or service clusters from

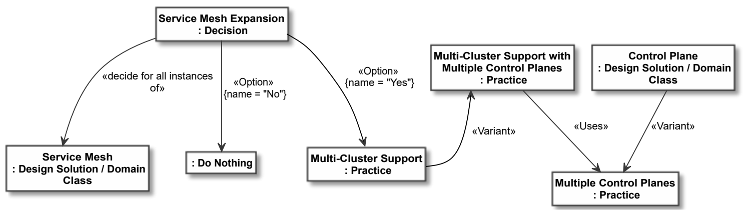
Service Mesh Expansion : Decision
«decide for all instances of»
«Option» {name = "No"}
«Option» {name = "Yes"}
Multi-Cluster Support with Multiple Control Planes : Practice
Control Plane : Design Solution / Domain Class
«Variant»
«Uses»
«Variant»
Service Mesh : Design Solution / Domain Class
: Do Nothing
Multi-Cluster Support : Practice
Multiple Control Planes : Practice

Fig. 4. Service Mesh Expansion Decision

the default configured for the service mesh. Proxy tasks generally can be implemented on the *Service Proxies* or in some cases alternatively on a *Front Proxy*. All solutions relying on a central service or on the *Front Proxy* are introducing a *single point of failure* (not repeated per case below). The decisions, options, and decision drivers for central services and proxy tasks are discussed below. In addition, we have found evidence for decisions that are needed for the basic functioning of the service mesh such as *policy distribution*, which we have not included in our catalog, as the user does not have to make a decision about them. Based on the services and tasks listed below we found evidence for many possible follow-on decisions such as support for *rate limits*, *quotas*, *circuit breaking*, *retries*, *timeouts*, *fault injection*, *dashboards*, *analytics*, and so on. We did not include those in our ADD model either, as the possible list of such higher-level services is excessive and will likely grow over time.

Service Discovery Decision In order to communicate in a service mesh, services need to locate others based on information like IP address and port number. Of course, this might be simply *hard-coded in each of these services*. If a service changes its address, fails or is unavailable for other reasons like congestion, then it becomes *not reachable* anymore. Then, there is a huge problem since all services code needs to be changed and the mesh needs to be restarted which impacts negatively *availability*. To resolve this issue, the *Control Plane* and *Data Plane* use *service discovery system* usually provided by platforms like Kubernetes for example. An alternative is using a central *Lookup* service [24]. The *distributed service discovery* option requires a consistency protocol and caching of discovery information, i.e. it is more *complex*. However, the lookup is local, thus it offers better *performance*. Without service discovery, the *manageability*, *changeabiltiy*, and *evolvability* of the service mesh would severely suffer.

Load Balancing Decision Service meshes, especially at scale, have to handle tremendous traffic loads which might overload services, increase their *latency* and decrease their *availability*. In order to avoid such a situation and maintain *scalability*, services are replicated and load is distributed over these instances by both the *Control Plane* and *Data Plane* using a *load balancing* algorithm. *Load balancing* can also be based on geographical location, especially in the case of service mesh expansion described in Section 4.4. If load balancing is used, the typical option is *Load balancing on the Service Proxies*. An alternative which offers *balancing loads for the whole ingress traffic* is *Load balancing on the Front Proxy*; this option offers less *fine grained control over the load balancing* than e.g. to balance per service cluster. Both solutions can also be *combined*, offering the benefits of both solutions but also increasing the *complexity*.

Custom Routing Decisions To manage cross-service and ingress communication, the *Control Plane* and *Data Plane* need to know where each packet should be headed to or routed; routing is usually configured on the *Control Plane* and enacted by the proxies on the *Data Plane*. In addition to such basic routing, the service mesh often offers *Custom Routing* options which can be based on URL path, host header, API version or other application-level rules for *control over the routing* in the mesh. Such routing rules can be dynamically changed, increasing the *flexibility* of the architecture. Custom routing can in follow-on decisions be used for extra tasks, a prominent one is to support continuous experimentation techniques such as *staged rollouts*, *A/B testing* and *canary deployment* (or not). The latter can help for more *controlled deployments* to production, which helps to minimize *deployment risks*.

Health Checking Decision In highly versatile environment such as service meshes, services go up and down unexpectedly which decreases *availability*. To overcome this issue, periodic *health checks* on services can be applied and e.g. mitigation strategies like service restarts can be applied. Health checks are usually performed by the service proxy and a central service collecting the information. Alternatively, simple health checks like pinging service proxies can also be done solely on the central service, but then *more complex health checking* is not possible. Of course another decision option is to not perform health checks.

Security-Related Decisions Communication in service meshes usually uses encryption based generated keys and certificates; if not used, the service mesh might be exposed to malicious traffic and manipulations, unless a key and certificate management service outside of the service mesh can or must be used. A simple option is using *API Keys* [26] and local key management. The alternative is to introduce a *central certificate authority*, residing in the *Control Plane*, that takes care of storing and distributing security keys and certificates to the *Data Plane*. This option is more *secure* than the other options and creates in large installations less *maintenance overhead* for managing various *API Keys* in the clients and service proxies, but it is also more *complex* than e.g. *API Keys*. Once authentication is handled, authorization needs to be considered. This can be achieved by setting up access control in the *Control Plane* or in the *Data Plane*. If we choose not to control access after authentication, then services are exposed to unintentional and unwanted modifications. *Security* is the most important driver in this decision; a solution on the data plane supports more *fine-grained control* but is more *complex* than a solution on the control plane. Using encryption in service meshes, usually based on mutual TLS, has to be handled at both ends; not using encryption means *security* is endangered. There are three decision options for *TLS Termination*: either we offer TLS termination directly in the service, at the *Front Proxy*, or – the most common option – in the *Data Plane*. The first option brings *boilerplate code* to the service which might also decrease its *performance*. The second option is only viable if the service mesh is in a private environment in which *internal unencrypted communication* is an option (or another encryption than the one used for communication with clients).

Collect Telemetry, Traces, and Metrics Decision To observe telemetry, traces and metrics in a service mesh, they first need to be collected. Otherwise, we have to access each of the services and upload this data manually. This is usually done by a *control plane service collecting data from data plane proxies*. With few services, we can choose not to *collect them centrally*. At *large scale*, this might make control and management tasks complex and *central features* such as dashboard or metrics are hard to impossible to build. Some telemetry might also be needed anyway for the functioning of the *service mesh itself*.

Multi-Protocol Support Decision In heterogeneous environments like service meshes, *multi-protocol support* is required. It helps to have a *unified API interface* that can be used by services using different protocols, which increases *interoperability* and *extensibility* of the service mesh. This can be *offered by data plane proxies* or *on the front proxy*, where the latter option offers less *fine-grained support* and is suitable if *the mesh uses only one protocol inside*. Of course, we might choose not to use this *API interface* and relieve the service mesh from the resulting processing *overhead*. Then, we need to add *boilerplate code* to services to support different protocols or suffer from *interoperability* issues.

5 Estimation of Uncertainty Reduction

There are many different kinds of uncertainties involved in making ADDs in a field in which the architect's experience is limited. The obvious contribution of our ADD model is that it helps to reduce the uncertainty whether all relevant, necessary and sufficient elements for making a correct decision have been found. Another kind of uncertainty reduction is the uncertainty reduction our ADD model provides compared to using the same knowledge, but in a completely unorganized fashion. We want to estimate this kind of uncertainty reduction here, following the approach described in detail in [26]. Here, we estimate the uncertainty reduction only for each individual decision. Please note that in most decisions combinations of options from different decisions need to be taken; but as many decisions in our ADD model have different decision contexts, this can only be calculated precisely for actual decisions made, not for the reusable decisions in the ADD model. But a consequence is that the actually achievable uncertainty reduction is much higher than the numbers below when decisions need to be made in combination. We calculate each number both for using our ADD model (denoted with $^{\oplus}$ below) and not using our model (denoted with $^{\ominus}$ below). Let DEC denote the decisions in our ADD model. For each, $d \in DEC$ there are a number of decision options OPT_d possible to choose for decision d. Finally, there is a set of criteria CRI_d that need to be considered when making a decision d.

Number of decisions nodes (ndec): Our ADD model represents each decision separately. So the number of decision nodes for a single decision d is always $ndec^{\oplus}_{d} = 1$. Without our ADD model, each decision option in the design space that is not *Do Nothing* is a possible decision node, and it can either be selected or not: $ndec^{\ominus}_{d} = |OPT_d \setminus \{\textit{Do Nothing}\}|$. Please note that, if a design solution has variants, OPT_d contains the base variant plus each possible variant.

Number of required criteria assessments in a decision (ncri): Our ADD model includes explicit decision criteria per decision and for all decisions described above all criteria are pre-decided in the sense that we have assigned a qualitative value *{++, +, o, -, --}* to it, represented in the range: very positive, positive, neutral, negative, and very negative. Thus the required criteria assessments per decision are one assessment per decision, $ncri^{\oplus}_{d} = 1$. Without our ADD model, we need to assess each criterion for each decision node (as we have no pre-decided choices): $ncri^{\ominus}_{d} = |CRI_d| \times |ndec^{\ominus}_{d}|$.

Number of possible decision outcomes (ndo): Our ADD model already models each decision option separately in $|OPT_d|$ including *Do Nothing* options, so $ndo^{\oplus}_{d}$ usually equals $|OPT_d|$ unless the design space allows explicit combinations of solutions as additional outcomes. For instance, in the decision on *managed ingress communication* the *API Gateway* can be combined with the base variant *Front Proxy*. Let the function *solComb*() return the set of possible solution combinations in the options of a decision; then $ndo^{\oplus}_{d} = |OPT_d| + |solComb(OPT_d)|$. The same is true in principle for the decisions made without our ADD model, but as the decision d is here split into multiple separate decision nodes $ndec^{\ominus}_{d}$ and without the ADD model no information on which combinations are possible is present, we need to consider any possible combination in $ndec^{\ominus}_{d}$, i.e., the size of the powerset of the decision nodes: $ndo^{\ominus}_{d} = |\mathcal{P}(ndec^{\ominus}_{d})| = 2^{|ndec^{\ominus}_{d}|}$.

Table 2 shows the results of the uncertainty reduction estimation. It can be seen that the number of decisions to be considered *ndec* can be in total reduced from 33 to 14, with an average improvement of 49.76% when using our ADD model. As all decisions have multiple criteria and when not using our ADD model no decision are pre-decided, the improvement for criteria assessments is higher: on average a 86.70% improvement is possible. Finally, the possible decision outcomes is improved from 96 to 47, with an average 32.59% improvement.

6 Discussion and Threats to Validity

We have studied knowledge on established practices in service mesh architectures, relations among those practices, and decision drivers to answer our research questions **RQ1-3**, respectively, with multiple iterations of open coding, axial coding, and constant comparison to first codify the knowledge in informal codes and then in a reusable ADD model. Precise impacts on decision drivers

Table 2. Uncertainty Reduction Estimation

Decision	$ndec^{\oplus}$	$ndec^{\ominus}$	**Imp.**	$ncri^{\oplus}$	$ncri^{\ominus}$	**Imp.**	$ndo^{\oplus}$	$ndo^{\ominus}$	**Imp.**
Managed Cross-Service Communication	1	3	66.67%	1	24	95.83%	4	8	50.00%
Managed Ingress Communication	1	2	50.00%	1	16	93.75%	3	4	25.00%
Traffic Control	1	5	80.00%	1	45	97.78%	6	32	81.25%
Service Mesh Expansion	1	2	50.00%	1	16	93.75%	3	4	25.00%
Service Discovery	1	2	50.00%	1	16	93.75%	3	4	25.00%
Load Balancing	1	3	66.67%	1	21	95.24%	4	8	50.00%
Custom Routing	2	4	50.00%	2	8	75.00%	6	10	40.00%
Health Checks	1	2	50.00%	1	6	83.33%	3	4	25.00%
Security	3	7	57.24%	3	24	87.50%	10	16	37.50%
Telemetry	1	1	0.00%	1	4	75.00%	2	2	0.00%
Multi Protocol Support	1	2	50.00%	1	16	93.75%	3	4	25.00%
Total	14	33		14	196		47	96	
Average Improvement Per Decision			49.76%			86.70%			32.59%

of design solutions and their combinations were documented as well; for space reasons we only summarized those in the text and did not show them in detailed tables. In addition, we estimated in Section 5 the uncertainty reduction achievable through the organization of knowledge in our ADD model. We may conclude that our ADD model (and similar models) has the potential to lead to substantial uncertainty reduction in all evaluation variables due to the additional organization it provides and pre-selections it makes. For individual decisions, mastering and keeping in short term memory the necessary knowledge for design decision making seems very hard for the case without the ADD model (see numbers in Table 2), but quite feasible in case of our ADD model. Our model also helps to maintain an overview of the decisions $ndec^{\oplus}$ and criteria assessments $ncri^{\oplus}$ in the combinations of contexts. Only the number of possible decision outcomes for the combination of multiple decisions seem challenging to handle, both in the $ndo^{\oplus}$ and $ndo^{\ominus}$ case. That is, despite all benefits of our approach, the uncertainty estimations show that a limitation of the approach is that when multiple decisions need to be combined in a context, maintaining an overview of possible outcomes and their impacts remains a challenge – even if a substantial uncertainty reduction and guidance is provided as in our ADD model. Further research and tool support is needed to address this challenge. As our numbers are only rough estimates, further research is needed to harden them and confirm them in empirical studies, maybe based on a theory developed based on such preliminary estimations.

While we believe generalizability of our results beyond the knowledge sources we have studied is possible to a large extent, our results are limited to those sources and to a lesser extent to very similar service mesh architectures. Most of the sources were public Web sources; there might be inhouse practices not reported to the public by practitioners not covered here. Some of the sources were from the technology vendors, which might have introduced bias; but this is mitigated to a certain extent as we considered sources from most major service mesh vendors. Our results are only valid in our set scope; we do not claim any form of completeness. Possible misinterpretations or biases of the author team cannot be fully excluded and might have influenced our results. We aimed to mitigate this threat by our own in-depth experience and by carefully cross-checking among the sources in many iterations.

7 Conclusions

We have performed in this paper a qualitative study in which we have studied service mesh established practices and proposed a formally defined ADD model. In total based on our findings, we modeled 14 architectural design decisions with 47 decision outcomes and 77 decision drivers. In our uncertainty reduction estimations we were able to indicate that the knowledge organization in our ADD model can lead to a significant reduction of uncertainty. We plan in our future work to combine our ADD model with other aspects of microservice design and DevOps practices, and empirically validate a theory based on the preliminary uncertainty reduction estimations. We also plan to validate our ADD model using real life case studies with field practitioners.

Acknowledgments. This work was supported by: FFG (Austrian Research Promotion Agency) project DECO, no. 846707; FWF (Austrian Science Fund) project ADDCompliance: I 2885-N33

References

1. Baresi, L., Garriga, M., De Renzis, A.: Microservices identification through interface analysis. In: European Conference on Service-Oriented and Cloud Computing. pp. 19–33. Springer (2017)
2. Charmaz, K.: Constructing ground theory. Sage (2014)
3. Coplien, J.: Software Patterns: Management Briefings. SIGS Books & Multimedia, New York (1996)
4. Eismann, S., Kistowski, J., Grohmann, J., Bauer, A., Schmitt, N., Herbst, N., Kounev, S.: Teastore: A micro-service reference application for cloud researchers. In: 2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion). pp. 11–12. IEEE (2018)
5. Envoy: Envoy is an open source edge and service proxy, designed for cloud-native applications, `https://www.envoyproxy.io/`
6. G. Glaser, B., Leonard Strauss, A.: The Discovery Of Grounded Theory: Strategies For Qualitative Research, vol. 3. Aldine de Gruyter (01 1967)
7. Gorton, I., Klein, J., Nurgaliev, A.: Architecture knowledge for evaluating scalable databases. In: Proc. of the 12th Working IEEE/IFIP Conference on Software Architecture (WICSA 2015). pp. 95–104 (2015)
8. Gupta, A.: Microservice design patterns. `http://blog.arungupta.me/microservice-design-patterns/` (2017)
9. Haselböck, S., Weinreich, R., Buchgeher, G.: Decision guidance models for microservices: Service discovery and fault tolerance. In: Proceedings of the Fifth European Conference on the Engineering of Computer-Based Systems. ACM (2017)

10. Hasselbring, W., Steinacker, G.: Microservice architectures for scalability, agility and reliability in e-commerce. In: Software Architecture Workshops (ICSAW), 2017 IEEE International Conference on. pp. 243–246. IEEE (2017)
11. Istio: What is Istio?, `https://istio.io/docs/concepts/what-is-istio/`
12. Jamshidi, P., Pahl, C., Mendonça, N.C., Lewis, J., Tilkov, S.: Microservices: The journey so far and challenges ahead. IEEE Software **35**(3), 24–35 (2018)
13. Lewis, J., Fowler, M.: Microservices: a definition of this new architectural term. `http://martinfowler.com/articles/microservices.html` (Mar 2004)
14. Lytra, I., Sobernig, S., Zdun, U.: Architectural Decision Making for Service-Based Platform Integration: A Qualitative Multi-Method Study. In: Proc. of WICSA/ECSA. pp. 111–120 (2012)
15. Morgan, W.: The history of the service mesh, `https://thenewstack.io/history-service-mesh/`
16. Morgan, W.: What's a service mesh? and why do i need one? (4 2017), `https://blog.buoyant.io/2017/04/25/whats-a-service-mesh-and-why-do-i-need-one/`
17. Newman, S.: Building Microservices: Designing Fine-Grained Systems. O'Reilly (2015)
18. Pahl, C., Jamshidi, P.: Microservices: A systematic mapping study. In: 6th International Conference on Cloud Computing and Services Science. pp. 137–146. Rome, Italy (2016)
19. Pautasso, C., Zimmermann, O., Leymann, F.: RESTful Web Services vs. Big Web Services: Making the right architectural decision. In: Proc. of the 17th World Wide Web Conference (WWW). pp. 805–814 (April 2008)
20. Richardson, C.: A pattern language for microservices. `http://microservices.io/patterns/index.html` (2017)
21. Selimi, M., Cerdà-Alabern, L., Sánchez-Artigas, M., Freitag, F., Veiga, L.: Practical service placement approach for microservices architecture. In: IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing. IEEE (2017)
22. Smith, F.: What is a service mesh? (04 2018), `https://www.nginx.com/blog/what-is-a-service-mesh/`
23. Truong, H.L., Gao, L., Hammerer, M.: Service architectures and dynamic solutions for interoperability of iot, network functions and cloud resources. In: 12th European Conference on Software Architecture: Companion Proceedings. p. 2. ACM (2018)
24. Voelter, M., Kircher, M., Zdun, U.: Remoting Patterns - Foundations of Enterprise, Internet, and Realtime Distributed Object Middleware. J. Wiley & Sons (2004)
25. Zdun, U., Navarro, E., Leymann, F.: Ensuring and assessing architecture conformance to microservice decomposition patterns. In: International Conference on Service-Oriented Computing. pp. 411–429. Springer (2017)
26. Zdun, U., Stocker, M., Zimmermann, O., Pautasso, C., Lübke, D.: Guiding architectural decision making on quality aspects in microservice apis. In: 16th International Conference on Service-Oriented Computing, ICSOC 2018 (2018)
27. Zheng, T., Zheng, X., Zhang, Y., Deng, Y., Dong, E., Zhang, R., Liu, X.: Smartvm: a sla-aware microservice deployment framework. World Wide Web **22**(1), 275–293 (2019)
28. Zimmermann, O.: Microservices tenets. Computer Science - Research and Development **32**(3), 301–310 (Jul 2017)
29. Zimmermann, O., Koehler, J., Leymann, F., Polley, R., Schuster, N.: Managing architectural decision models with dependency relations, integrity constraints, and production rules. J. Syst. Softw. **82**(8), 1249–1267 (2009)