MADIS Doctoral School
Inria Centre at the University of Lille
CRIStAL — Spirals research team

Concern-Oriented MicroService Architecture: Language, Library, Toolbox, and Evaluation

Hugo **MONFLEUR**

Thesis submitted for the degree of doctor of philosophy in computer science
Defended on 28 November 2025

Supervisor

Philippe Merle	Research Director	Inria

Reviewers

Thomas Ledoux	Professor	IMT Atlantique
Christian Perez	Research Director	Inria

Examiners

Anne Etien	Professor	University of Lille (Jury President)
Vania Marangozova	Professor	University of Grenoble
Anthony Cleve	Professor	University of Namur

École Graduée MADIS
Centre Inria de l'Université de Lille
CRIStAL — Équipe de recherche Spirals

Architecture à Microservice Orientée sur les Préoccupations : Langage, Bibliothèque, Boîte à Outils et Evaluation

Hugo MONFLEUR

Thèse présentée en vue de l'obtention du grade de docteur en informatique
Soutenue le 28 novembre 2025

Directeur

Philippe Merle	Directeur de Recherche	Inria

Rapporteurs

Thomas Ledoux	Professeur	IMT Atlantique
Christian Perez	Directeur de Recherche	Inria

Examinateurs

Anne Etien	Professeure	Université de Lille (Présidente du Jury)
Vania Marangozova	Professeure	Université de Grenoble
Anthony Cleve	Professeur	Université de Namur

Abstract

The microservice architectural style has risen to a prominent place during the last decade in synergy with the development of the cloud. Those two aspects of software architecture work in symbiosis with the distributed and independent nature of microservices permitting central requirements of cloud hosted applications such as scalability, flexibility and efficiency in resource utilisation to be fulfilled while the tendency to host applications on the cloud has given its full relevance to the microservice architecting style. Even though microservice architecture has spread by virtue of its undeniable advantages, it has brought new difficulties equally consequences of the source of its benefits. The multiplication of independent executable entities is an hindrance to global representation of the system which coupled to the independence of development teams can lead to an increasingly obscure and pointlessly complicated state. To tackle this challenge, software architects can rely on Architecture Description Languages (ADL) whose aim is precisely to represent the architecture of a system. Multiple ADL propositions have been made to represent microservice architecture about which we identified two main shortcomings. Firstly they share a common model that doesn't make overarching abstract structures such as design patterns first class citizens of the languages. Secondly their ultimate focus is to generate procedural execution code which goes against the fundamental technology agnosticism of the microservice style but also condemns the ADL to be irreversibly desynchronised with the actual application since the generated code needs completion and adaptation. To answer those unsolvable limitations we propose a new microservice ADL model and language, named Concern Oriented Microservice Architecture (COMSA) and its associated toolbox. We use the Concern concept to semantically anchor a structure encompassing microservices made first class citizen of the language thus making architecture description a necessity by design. That orientation is supported by compilers producing code executable without any modification thus making COMSA an executable language. This is made possible by choosing declarative deployment languages as target languages which permits not to consider the order of instructions in compilation and leave complete freedom of choice to the developers in terms of microservices implementation technology. Through rewriting of a dataset of Docker Compose microservices applications in the COMSA language, we established that on average application used tree times less properties indicating a major increase in coherence and supporting the qualitative benefits we describe in terms of understandability, error resilience and maintainability.

Résumé

Dans les dix dernières années, le style d'architecture à microservices a pris une place majeure dans la conception des applications. La croissance de son importance s'est faite en symbiose avec le développement du *cloud*, la nature distribuée et indépendante des microservices trouvant sa pertinence en permettant aux applications d'exploiter les possibilités de l'infrastructure telles que la mise à l'échelle, la flexibilité et l'efficacité dans l'utilisation des ressources. Bien que le style d'architecture à microservices doit sa large adoption dans l'industrie à ses indéniables mérites, il a aussi apporté de nouvelles difficultés consubstantielles à ses avantages. La multiplication d'entités exécutables indépendantes rend la représentation du système difficile, et le fait de permettre l'indépendance des équipes de développement, bien que produisant des gains considérables dans les délais d'intégration, créé un risque accru de dérive vers des systèmes inutilement complexes et obscurs. Pour limiter ces conséquences néfastes, les architectes logiciels peuvent s'appuyer sur des Langages de Description d'Architecture (ADL). Par l'étude des ADL spécifiques aux applications à microservices existants, nous avons identifiés deux principaux manquements qui sont le partage d'un métamodèle ne faisant pas des structures architecturale abstraites des citoyens de première classe dans le langage et d'avoir pour finalité de produire du code métier procédural ce qui, d'une part, va contre l'agnosticité technologique qui est au cœur de l'esprit des applications à microservices, et d'autre part produit une inévitable désynchronisation entre la description architecturale et l'application puisque le code généré demande d'être adapté et complété. Pour répondre à ces limitations, nous avons conçu un nouveau métamodèle pour les ADL microservices ainsi qu'un language l'implantant nommé Concern Oriented Microservice Architecture (COMSA) accompagné d'une boîte à outils. Nous proposons ainsi une nouvelle manière de concevoir la description des applications à microservices, en utilisant le concept de préoccupation pour ancrer sémantiquement une structure syntaxique faisant de la description architecturale une nécessité de conception. Cette proposition est soutenue par des compilateurs produisant du code exécutable sans modification faisant de COMSA un langage exécutable. Par la réécriture d'un jeu de données d'applications décrites via le langage Docker Compose dans le langage COMSA, nous avons établi qu'une application écrite en COMSA demandait en moyenne trois fois moins de propriétés pour une description équivalente. Ce résultat indique immédiatement un gain considérable en cohérence et vient corroborer les améliorations qualitatives en intelligibilité, tolérance aux erreurs et efforts de maintenance.

Acknowledgements

I would first like to thank Philippe Merle for giving me this opportunity and for supervising this thesis, which owes much to his scientific culture, his experience, and his ability to identify relevant recent technological developments. I would like to thank Thomas Ledoux and Christian Perez for the careful reading of this manuscript and their detailed reports, as well as Anne Etien, Vania Marangozova, and Antony Cleve for their participation in the defense and the quality of the discussions. I would also like to thank Inria as a whole for providing me with insight into the world of research, and in particular the Spirals team, whose members are committed and welcoming and who warmly received me. I thank the members of the Scaler project who accompanied me throughout these three years. I would like to thank the University of French Polynesia, and in particular its president, for allowing me to resume my studies—initially for two years, but which ultimately concluded eight years later. Above all, I would like to thank Virginie, for everything, and my children Thibault, Antoine, and Agnès, for being there.

Hugo MONFLEUR
Lille, January 2026

Table of contents

List of figures

List of tables

Chapter 1

Introduction

1.1 Motivation

The microservice architecting style has risen as one of the main programming and execution paradigms in the last decade with a majority of enterprises already having and continuing to incorporate the technology in their systems [11] and is expected to maintain its growth in absolute terms as well as in market share in the years to come [44] [82] [83] [38] [45]. It has delivered many of its announced benefits regarding modularity, technology freedom, scalability, fault tolerance or development independence. The source of those considerable gains is first a technological revolution in the conception of applications structures. Applications were first considered as single executional unit, from which the term "monolithic application" comes from. They could logically be decomposed into different parts, but each of those parts are in the end indissociable, having to share the same execution environment in terms of software as well as in term of hardware. The idea of microservices is to have an executional functioning mirroring the logical design of applications. Each logical part of the application becomes its own process, able to run independently from other parts using different software and potentially running on different hardware. Those independent components interact with each other, usually through network communication, to act together as a coherent system identical from an exterior point of view to a monolithic application.
Even though it has resolved most of the challenges modern application hosting and development poses, it has also renewed the ancient domain of software achitecture bringing new difficulties while solving old ones. Through bounded context and dedicated execution environment microservices has removed most of the coupling difficulties that were a constant concern in monolithic applications. On one hand, microservices function as a black box to their counterparts and as such they are totally independent in their implementation allowing small teams to practice continuous integration and development as long as the contract they have with the other parts of the application are respected. On the other hand, that new level of autonomy, distribution and evolvability, all coming

from the execution independance of microservices, has brought new architectural risks [61] limitable only by a considerable supplementary burden at design time as well as during the application life cycle.
Mitigating this problem relies mainly on thoughtful decisions and even thought tools and analysis means have been proposed, the lack of solutions globally adopted hints to a fundamental problem that cannot be ideally solved. The first and most important step in making those decisions is to dispose of a clear view of the system. To that effect we considered the specific languages describing the applications in their entirety, thus being by definition Architecture Description Languages (ADL). Looking at the existing solutions we concluded that they did not provided a satisfying solution due to a common and unfited model. We then decided to conceive a new microservice Architecture Description Language based on a simple but radically different model.

1.2 Research questions

- **RQ#1: On the current state of microservices ADLs:**
  - What is the current state of microservice ADLs?
  - Do they adequatly describe microservices-based applications?
  - Which gains do they bring?
  - What are their limitations ?
- **RQ#2: On a new model for microservice ADL:**
  - Which model would fit microservice applications description?
  - What gains can be expected?
- **RQ#3: On the semantics for a new microservice ADL:**
  - What are the objects that would be represented to form the semantics of a new microservice ADL?
  - How are those objects represented in real world applications ?

1.3 Contributions

1.3.1 The State of The Art

We first have identified the available Architecture Description Languages specifically oriented toward microservice-based application. Then we have gathered a list of characteristics and features used to qualitatively evaluate ADLs in general and applied those criteria to the identified languages. As a general conclusion we answer negatively to RQ#1 with

the observation that architecture is not adequatly described and that microservices ADLs focus more on the component description than on the applications topology. Two main flaws in our view are the source of this situation. First, most of the languages are based on a model that instead of having architectural constructs at the root level of the descriptions have the microservices. Because of that architectural links are, if expressed, relegated as in line attributes of microservices which does not provide and immediate and clear view of the application structure. Second, most microservices ADLs focus ultimately on business code and generally on generation which poses multiple problems whose principal one is that in all cases code produced has to be modified to be executable, an immediate and ever growing divergence between the runnable code and the description happens making the later incorrect and ultimately useless. That focus also goes against one of the fundamental benefits of the microservice which is to allow technology freedom in service implementation as the burden to maintain code generation for a single language is considerable, all the more for multiple languages and at best delayed for latest modifications, new and less spread technologies.

1.3.2 A new model for Microservices Architecture Description

The most fundamental contribution is the proposition of a new model for microservice application architecture description answering RQ#2. We thought of a solution to remedy to the limitations we identified in the existing solutions and decided to introduce a descriptive structure as first class citizen of the language fitting to express the architectural structures. The landscape of what can be considered an architectural structure is wide and hard to encompass and we quickly realized that limiting it to architectural design patterns was not appropriate to represent the point of view of the architects. Answering part of RQ#3 we decided on utilizing the Concern concept which is an ancient idea in informatics that aims to segment descriptions in multiple points of views that represent the architect vision. The formal materialization of that abstraction is to link sets of components to sets of properties shared by the former and that pertain to a certain concern of the architect relative to the application. While proposing a radically new form of description we ensured that it was compatible with the previous model by being an extension that still allowed to express microservices at the root of the description alongside Concerns. In addition to the formal compatibility it permits users to transition slowly incorporating properties from services to Concerns and provides a focus at design time by having a reflection on properties not included in patterns and their architectural significance. The modest but existing overload of introducing a new abstraction is largely countered by the expected gains in coherence, maintainability and error avoidance since multiplication of identical properties is a common and unfortunate phenomenon in microservice-based applications descriptions that can be almost entirely removed by their unification in really unique properties in Concern contained property sets. The question of recomposition in

the perspective of execution of Concerns based descriptions to functionally equivalent ones but without the supplementary Concern abstraction is an historical limit of Concern oriented programming that we solved by describing our model possible implementation as necessarily declarative thus removing the question of order of properties in the produced description. From that we identified deployment files as the best candidates because of their commonly declarative nature but also because of their technological agnosticity relatively to services implementation which fits the microservice spirit and focuses on the topology rather than the technological specificities.

1.3.3 Concern Oriented MicroService Architecture: A Concern-based language and associated toolbox for microservice application description and deployment

Having established a new model for microservice application we went on to propose a concrete syntax for our language model as well as an associated toobox to make it executable. We used Pkl, a new programming language blending object oriented and functional programming and focusing on configuration to establish the concrete structure of description files as well as the definition of objects whose central element is the Concern concept implemented as a class. To make the language usable we developed a series of tools among which the compilers are the most essentials by allowing translation to immediately executable Docker Compose files and Kubernetes manifests making de facto COMSA an executable language. We completed the toolbox with two representation tools. The first produces a view of concerns in the form of hypergraphs encompassing services and allowing for an high level view of the system and spotting inconsistencies. The second produces a view of the communication links and deployment dependencies expressed in the description files. Other tools have been developped in order to analyse the architect descriptions and detect possible inconsistencies.
In order to help users structuring their applications, we established a library of concerns which fueled a reflection on Concerns types that led us to organise our Concern tree between property centric and service centric concerns. That original distinction appeared to have intrinsic relevance not only as a precision to the answer to RQ#3 but also as an underlying characterization of the taxonomy of the different entities in microservice architectural element. To give an evocative example, the restart policy for an application is centered on a property often shared by all microservices while the communication pattern is centered around a particular microservice. Another original characterization that emerged was the observation that technical services distinguish themselves from business services by being at the center of a Concern, avoiding a tedious listing of the multiple technical types and technologies.

1.3.4 Numeric Evaluation of COMSA and Microservice Application Dataset Analysis

While we were confident in our arguments in the advantages provided by a concern oriented ADL for microservice application description, we wanted to establish its relevance through numerical data. To that effect we used a curated dataset of microservice applications and translated 21 Compose descriptions in the COMSA concrete syntax. We decided on Compose description because of its representativity of the common microservice ADL model and because it has very little language specific structures. We then determined a metric to compare both kinds of descriptions and decided on properties understood as an instance of an unique association of a value to an attribute. Through this process we arrived at a our main scientific result showing that on average it took tree time less properties to express description at least equivalent, meaning that any property assigned to a service in the original description was also assigned explicitly or implicitly to the main service. Indeed, specific concerns, particularly design patterns, allow to not explicitly mention a property but assume its presence and it being generated at compile time. Through that process the COMSA language goes over the possible gain in property reduction on average by a factor of 0.85 and even expresses more than what was in the original description file on average doubling the number of expressed dependencies.
Using our COMSA dataset to answer RQ#3, we used it to analyse the concern composition of existing applications to extract widely spread architecture deficiencies such as the limited share of application having an explicit restart policy. Based on our observations we propose the idea of characterizing microservice applications through sets of Concerns. Having found the API gateway pattern as being the most spread across the dataset, we found that the abstract modular application type combining API gateway with one of two forms of communication pattern (service registry vs messaging) and one of two forms of data storage and access patterns (dedicated vs shared) covers 38% of the dataset.

1.4 Outline

We start with Chapter 2 by contextualising the microservice architecture style in software archictecture development going from monolithic architecture to microservice and passing by service oriented architecture. For each style we describe their fundamental structure model and the limitations that lead to the appearance of the next style in the hope that they would be overcome. Arriving at microservices we conclude that when applied to appropriate situations, it brings most the gains it was aiming at but that its fragmented nature also brings some new pains that can be mitigated with a clear view of the system and that the best tools for providing it are architecture description languages.

In Chapter 3 we present the different criteria to categorize and evaluate Architecture

Description Languages. We then go on with a presentation of the identified exiting microservice ADLs before applying the evaluation grid we established. We conclude this chapter by expressing our dissatisfaction with the available languages because of their focus on describing the components rather than the architecture thus missing the original aim which then limits the tools proposed around the language.

Chapter 4 is the heart of this thesis as it presents our original proposition to remedy the unsatisfying state of the art we have observed in the previous chapter. It first presents the underlying model we put forward as able to correctly describe microservice architecture which is an extension to what we identified as the common model to other microservices ADLs. It introduces a new structure as first class citizen that links sets of microservices and sets of properties. Through it we can unify scattered properties, thus reducing their numbers and increasing coherence of the system. We then introduce the Concern concept as the appropriate instantiation of this new structure because of its malleability that allows to encompass any aspect of microservice-based applications. The most part of this chapter consists in the main contribution of this thesis and is the exposition of the core of the COMSA language and the description files structure giving a concrete syntax to the Concern concept thus instantiating our new model. Doing so we present the gains allowed by the use of this new model.

Chapter 5 presents a library of concerns we have established and its implementation as classes inheriting from the Concern class. Aside from providing a large selection of examples extracted from the rewriting of a dataset of Docker Compose descriptions of applications, we introduce the fundamental distinction between Concerns based on a central properties and Concerns based on a particular microservice. We conclude from our analysis that property centric concerns are to be integrated into service centric concerns as much as possible and identify which categories of property centric concerns are legitimate to exist by themselves.We also provide a simple but meaningful syntactic characterization of technical microservices as being at the center of a pattern.

Chapter B presents the toolbox associated with the COMSA language. Its most important tools are the compilers allowing to translate COMSA descriptions of microservice application to functionally equivalent and immediatly executable Docker Compose and Kubernetes descriptions. That makes the COMSA language an executable language as no modification to the produced files are needed and thus an alternative to existing solutions for deployment. Other notable tools are representation tools allowing to produce graphical representation of the system, error checkers indicating architectural deficiencies and analytic tools providing metrics on applications based on their descriptions.

Chapter 6 presents the results of the application of the COMSA language to a set of

microservice applications whose original description files were done using the Docker Compose Language in Yaml format. The main result is that on average an application can be described in COMSA using tree time less properties than with Docker Compose. It directly confirms the coherence gain provided by using a concern-based language. The second part of the evaluation contains an analysis of the repartition of identified concerns in the dataset. Emerging from an individual analysis of concern repartition, we propose a characterization of microservice-based applications through tuples of instantiated concerns. In particular as we observe the API gateway pattern as the most spread in the dataset and we propose a combinatory characterization of the patterns function of the storage topology (shared or scattered) and communication choice (synchronous vs asynchronous).

Chapter 2

Emergence of Microservices

Microservice architecture is a paradigm that has emerged in the last decade and has evolved as one of the most considerable software architectural style. As such, it can be defined positively but having emerged from the industry as a response to the existing difficulties the prior paradigms encountered there is some value in looking at what was until then the dominating styles.

2.1 From Monolithic Applications to Microservices

2.1.1 Monolithic Applications

As the name indicates monolithic applications are to be considered as units of execution where all the application logic is being expressed in a single chosen language through its specific concepts with no general necessity to avoid coupling. As Dragoni puts it in [25] "A monolith is a software application whose modules cannot be executed independently." Because of that the physical support for the execution of a monolithic application is necessarily a single machine and scaling has to be done horizontally, by replicating the whole application [64].

2.1.2 Monolithic Applications Shortcomings

The nature of monolithic applications is the source of multiple technical issues among which we can distinguish those that are relative to development and those that are relative to deployment and operation.

On the development front we note that due to their monolithic nature, they grow to become large programs [65] making them "difficult to maintain and evolve" [25]. For the same reason multiple modules involve multiple libraries and because of that "adding or updating libraries results in inconsistent systems" [25] or imply important and costly code updates [65]. Finally having all parts of the code linked together implies a "technology

lock-in for developers, which are bound to use the same language and frameworks of the original application" [25] [65].

On the deployment and operation side, the necessity of a unique deployment environment for the entire application forces a "one-size-fits-all configuration" that implies having to arbitrate about a global environment in which different modules with different requirements will live. As the application has at its root a single process, any modification or failure implies a complete reboot provoking considerable downtime. The same reason makes horizontal scaling as coarse-grinded as imaginable as it is not possible to scale parts of the application but only to replicate it entirely [25] [65].

The implications of these limitations are not only technical but reflect on the developing team organization. As it is not possible to dissociate the execution of the different parts of the code, any modification has to be validated by a large proportion of the team or at least by some central authority. Indeed any default could potentially render the entire system unusable. Because of that decisions are heavily centralized and necessitate the consultation of a large number of people to ensure that the changes will not have a problematic effect on their area [25] [65].

2.1.3 Services

Service Oriented Architecture To tackle the limitations of the monolithic applications, a new paradigm has emerged in the early 2000s called Service Oriented Architecture [59]. Put simply the idea is to decompose the "classical" monolithic application in multiple independent services modeled on real-world business activities. The inner workings of one service are unknown to one another and the interaction is done through exposed interfaces that services agreed upon in documents called "contracts" that define the operations provided and the means of communication to use them.

Those service interfaces are commonly exposed through Application Programming Interfaces (APIs), which define how services can be invoked by external clients or other services. Historically, Service-Oriented Architecture relied heavily on standardized protocols such as SOAP, often coupled with formal interface descriptions, in order to ensure interoperability across heterogeneous systems. In contrast, microservice architecture has largely favored lighter-weight communication styles, most notably RESTful APIs, which emphasize stateless interactions and resource-oriented design. More recently, alternative API paradigms such as GraphQL have emerged, offering increased flexibility in data querying at the cost of more complex API management.

The core novelty of SOA is then that while having the same behavior from the external user point of view, instead of having a big, one process, application as it is the case in monolithic architecture, the application will be a set of independent processes acting as one. Migrating from monolithic architecture, that can be done through splitting one

application into services or through reuniting different applications into one and affecting replicated functionalities to a particular service.

The legacy of SOA This somewhat technical, description of SOA could fit Microservice Architecture as well, and considered from a high-level perspective, Microservice architecture is indeed a particular instance of Service Oriented Architecture.

While fundamental the technical aspect of SOA is not its main contribution but instead a consequence to its main destination which is to be a design paradigm. It aims to realize the separation of business concerns through the concept of service using the following set of principles [28]:

- **Standardized Service Contract:** document indicating how to interact with a service.
- **Service Loose Coupling:** Aiming to have on one front minimal dependency between clients and the service contract and on the other front minimal - ideally none - dependency between the service contract and the concrete implementation.
- **Service Abstraction:** Expose as little as possible of service to its environment to allow for loose coupling.
- **Service Reusability:** Design services capabilities while being agnostic to functional contexts allowing for multi-purpose logic.
- **Service Autonomy:** Designing services having in mind "isolation levels and service normalization" to allow for availability, reusability, and scalability.
- **Service Statelessness:** Aiming at minimal statefulness to increase availability and scalability.
- **Service Discoverability:** Aiming for service accessibility and usability to be minimally dependent on the environment to enable maximum reusability.
- **Service Composability:** Design services so that they can be reused as parts of future compositions.

Here again, all those fairly general principles are central to Microservice Architecture design.

SOA vs MSA As we have seen MSA is not distinguished from SOA through its core technical mechanism or its principles as a design pattern. Moreover, MSA can be considered as a particular style of SOA.

However, SOA cannot be reduced to these general principles as it has existed as a real practice that MSA used as a basis to distinguish itself from, either in opposition or as a

precision. In its 2015 book [84], Richards notes the following distinctions between SOA and MSA :

- **Service characteristics:**
  - **Service Taxonomy:** MSA distinguishes only between functional services and infrastructural services. Functional services provide application-specific business functionalities while infrastructural services provide technical functionalities that are nonspecific to a particular application.

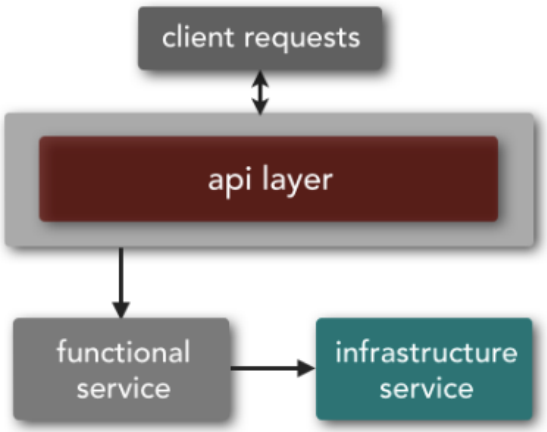


Figure 2.1: MSA taxonomy [84]

SOA has a much more developed taxonomy with four different types that can be ranked in levels of decreasing abstraction and increasing specificity. Business services are abstract services, void of implementation and define business operations at enterprise level. Enterprise services are concrete services implementing business services, they are usually "shared across the organization". Application services provide specific operations to enterprise services that are not business operations but are used to resolve them. Lastly, Infrastructure services provide non-business functionalities used to run the system.

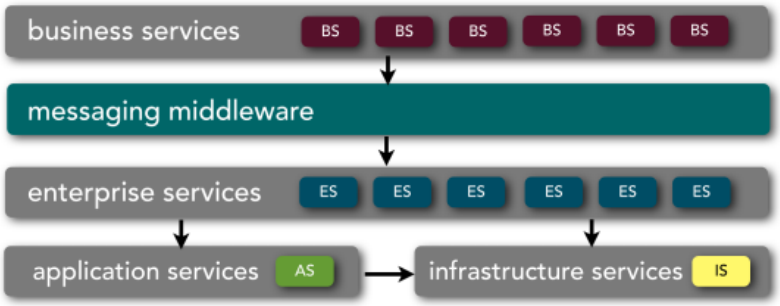


Figure 2.2: SOA taxonomy [84]

  - **Service Ownership:** In SOA service ownership matches service taxonomy with separate owners for separate types. An important similarity with Monolithic

applications is the distinction between business service teams and infrastructure service teams. The negative consequences of that separation are the same as in Monolithic applications development i.e. each decision implies the coordination of an important part of the organization.

  - **Service Granularity:** SOA does not give particular guidelines relative to service granularity while MSA aims for one distinct capability per microservice. In practice, SOA services are akin to monolithic applications going "from small application to very large enterprise services." Regarding Microservices it is to be noted that the service granularity problem is then transmitted to the capabilities granularity problem.

- **Architecture Characteristics:**

  - **Component Sharing:** SOA tends to favor a "share-as-much-as-possible" approach grouping resources in shared services used by as many other services as possible and thus clarifying subtle differences in shared concepts by removing functionality duplication but also increasing coupling by centralizing service dependencies. MSA takes the opposite approach namely "share-as-little-as-possible" favoring repetition over coupling.
  - **Orchestration vs Choreography:** SOA tends to favor orchestration of services i.e. having a central component controlling the general workflow while MSA is better suited for choreography, a style where the workflow is not centralized. In SOA, by design, a Middleware component (such as an Enterprise Service Bus) manages communication with the different enterprise services under which choreography can happen. The component responsibility can vary but it often manages message transformation so that a service can interact with another without knowing the precise input it needs. That allows for multiple communication protocols in an SOA application. Such a component is not fundamental in MSA where services must know how and where to request needed resources. Because of that the communication protocol must be unified throughout the entire MSA application and each microservice must know the exact contract of the other microservices it interacts with. It is to be noted that recent developments in microservices have brought orchestrating components e.g. Kubernetes.

2.1.4 Microservices' Solutions to Monolithic Applications Shortcoming

Microservices are meant to solve or at least mitigate the difficulties encountered by monolithic and service-oriented architectures. The gains that style has brought can be classified into different categories [90]:

- **Design time:** The usage of "bounded contexts" [30] [63] allows separation and encapsulation of the concepts of an application in smaller entities. This independence, with only some points of contact needing mutual understanding, brings easy governance, flexibility as well as it allows the application to be fault tolerance by design and permits it to be cloud native [90]. The particular nature of microservice applications has led to the development of specific design patterns [87] that either bring positive gains or at least mitigate the particular difficulties linked to microservice architecture [90].

- **Development:** The department in which microservice architecture shines the most relatively to its alternatives. The focus on loose coupling permits the independence of teams in development and in choosing the technology stack. Also, the encapsulation of simple logic in microservices allows to easily integrate new members to the projects [90] [102] [25].

- **Operation:** Loose coupling of microservices allows their independent deployment whether initial or as the application is online. Their deployment is made easy by the widely used container technology and its support by providers. Once online, they are fault tolerant in the sense that they can continue to run even if neighboring microservices are not which implies that their reboot is independent from the rest of the architecture [90] [102] [25].

  The biggest operation gain is the ability to scale microservices independently thus allowing for fine-grained scaling while the duplication of large components in the case of service-oriented architecture and of the whole application with monolithic architecture is required. Because of that microservices applications can provide better reaction time due to reduced deployment time and resource-saving due to the limitation of replicated microservices [90] [65] [25]. It is to be noted that those gains are relative to the ability to identify the microservices in need of scaling in a given situation.

.

We have seen that microservice architecture bring substantial gains relatively to preceding architecture styles. Those gains all come in one way or another from the independence microservices have from each other though being part of the same application. Unfortunately, those gains come with new difficulties.

2.1.5 Microservices Limitations

Microservices as a trade-off While microservices are a promising idea that can solve or alleviate some difficulties encountered in monolithic or service-oriented architectures, they have also some considerable drawbacks. Some of them are fundamentally linked to the nature of microservices, mostly due to the difficulty of using microservices appropriately.

The same classification used for the "gains of microservices" can be applied to the "pains of microservices" [90]:

- **Design Time:** While the technical foundation of microservices allows for the independence of microservices by design it also implies a global view of the application so that the independence given to teams does not mean harmful disorganization. The original idea of having one capability per microservice is tempting but often unclear or inefficient. Because of that finding the appropriate granularity for microservices is a difficult and risky task at design time [90] [35] [41]. The complexity of the architecture can also grow out of hand either by poor original segmentation design (such as having the above-mentioned inappropriate granularity) or because of unchecked growth in the creation of microservices. Managing that aspect, particularly when having to consider other difficulties inherent to microservices architecture such as security considerations or latency risks, can be challenging [90] [35]. Security is a particular concern that requires more attention than it does with monolithic or service-oriented architectures since the fact that every microservice exposes an API increases the surface of attack upon which the mandatory access control is rendered more difficult due to the heavily distributed nature of microservice applications [90]. Lastly while modifying the architecture, the independence of microservices implies the necessity of backward compatibility for other microservices to be able to realize their operations thus creating a risk of endpoint proliferations [90] [35].

- **Development:** While Development is the department where microservices shine the most, two aspects are made more difficult by the distributed nature of microservices: data storage (consistency, distributed transaction, heterogeneity, query complexity) and testing (performance and user experience) [90].

- **Operation:** Resource usage is a major concern in microservice architecture. Since intra-application communications go through it, network usage is where microservices have the biggest impact compared to other architectural styles [90] [35]. A lesser concern is the usage of compute resources that can be intensified due to the multiplication of runtime environments [90]. While it is generally possible to avoid single points of failure, cascading failures can still happen, service coordination and location are challenging causing difficulties in problem location that are increased by the scattering of logs in the application [90].

  It is to be noted that most of the difficulties contained in microservice architecture come from their distributed nature and independence from each other. If those difficulties can often not be entirely resolved, they can be alleviated at the cost of much work invested in the consideration of the architecture, but also, and much more easily, made worse when adopting inappropriate architectural decisions.

Microservices only mitigating Monolithic Applications Problems We have seen that the fundamental aspect of microservices being independent, distributed, entities collaborating loosely is a way to answer some unsolvable challenges of monolithic and service-oriented architecture. We have also seen that this very fundamental aspect of microservices is at the root of other difficult challenges. Telling which architecture style is better in general has no meaning but even in a particular context where an application could benefit from microservice architecture overall, what tip the scale is not the choice of that style but the architecture design for which no single general rule can be theorized other that having thoughtful consideration for all the factors involved.

But if such consideration is already a considerable challenge when building an application from scratch, it is made much more difficult when modifying an existing, on-line, application due to the independence of teams which is also one of the main appeal of microservice architecture. One way to keep track of the global picture for local actors is to share a map of the whole application expressed through an Architecture Description Language (ADL).

2.1.6 Broader Foundations of Microservice Architecture

While Service-Oriented Architecture constitutes the most direct conceptual predecessor of Microservice Architecture, it is not the only paradigm that has contributed to its foundations. Earlier work on distributed application models and component-based software engineering laid essential conceptual and technical groundwork that microservice architecture inherits, refines, or reinterprets. Distributed application models such as client–server architectures, multi-tier systems, and middleware-based systems introduced the decomposition of applications into interacting runtime entities deployed across multiple nodes, together with fundamental concerns such as partial failure, latency, coordination, and consistency that remain central in microservice-based systems [13]. In parallel, component-based software engineering promoted the design of systems as assemblies of reusable, composable units with explicit interfaces and well-defined contracts [93]. While components were often conceived as deployable units within a single application or platform, frequently sharing a runtime environment, deployment lifecycle, or memory space, microservices extend these ideas by enforcing stronger isolation boundaries, independent deployment, and decentralized ownership. From this perspective, Microservice Architecture can be understood not as a radical break from prior paradigms, but as the convergence of distributed systems engineering, component-based modeling, and service-oriented design, enabled by modern cloud-native infrastructure and operational practices [25]. Beyond communication styles and service decomposition strategies, recent execution models have further extended the microservice paradigm by modifying the way services are deployed and operated.

Serverless Computing It is worth mentionning serverless computing, a new development paradigm which manifest itself mainly in the form of Function as a Service (Faas). The novelty in this development paradigm is the complete abstraction of the infrastructure going as far as not having to consider the environment in which the functions are executing. The infrastructure is taken care of by a provider to which only the function code is transmitted through the network. The provider then manages the function execution before communicating its result. To support *could functions* other functionalities are provided, such as means of data persistence, which constitutes the other aspect of serverless computing, namely Backend as a Service (BaaS) [53] [62].

Its main appeal comes from the extreme elasticity relatively to scaling and the financial benefits linked to it since users only pay for the execution time of functions. Infrastructure abstraction is also a strong argument for serverless computing [62]. In those regards it maximizes some of the advantages provided by microservice architecture.
However it is also inclined to aggravate some of the risks associated with microservice architecture. Latency is much more acute concern both because of the network communication time and of the startup time for the containers that support cloud function executions (that roughly live only for the lifetime of the function invocation). Because of that last aspect granularity design is an even bigger concern. Parallelism is also difficult in a serverless context since function can only communicate through data storage in persistent storage which is costly in terms of latency and network ressources [62].

2.2 Conclusion

The emergence of microservice architecture has profoundly modified the conception of application structure, lifting some of most ingrained limitations that previous paradigm carried at their core. But the metamorphosis of the applications inner workings from executional monolith of intertwined functions to constellation of independent processes produced new challenges among which the need for careful design through original structural patterns can be considered the most critical to avoid pitfalls. Since no universal method for designing microservice based application is to be hoped for we can only strive to help architects with the appropriate means to support the careful reflection that the microservice paradigm requires. To that effect, we dedicate the next chapter to the analysis and evaluation of what is the most fundamental tool for such a task: Microservice Architecture Description Languages.

Chapter 3

Microservice Architecture Description Languages: State of the Art and Research Challenges

3.1 ADLs in general

General Definition of ADLs The name Architecture Description Language first appears to be self-explanatory. As does any formal language, an ADL has a syntax that needs to be respected for a description to be correctly coupled to a semantic that relates to an application architecture [9].

What's unsatisfying with such a simple description is that it encompasses much more than what we want to cover. After all, any programming language has a strict syntax and obviously contains the architecture it defines. What's expected from an ADL is, if not an abstraction, at least a global view. As Vestal puts it "*An ADL for software application focuses on the high-level structure of the overall application rather than the implementation details*" [101].

Trying to get to a more precise definition of what an ADL is we can take a look at ISO/IEC/IEEE 42010:2022, Software, systems, and enterprise–Architecture description [47] that defines Architecture Description Languages as *"means of expression, with syntax and semantics, consisting of a set of representations, conventions, and associated rules intended to be used to describe an architecture*". Here again, the definition lacks an identifying factor to set ADLs aside from other languages. Indeed it is difficult to identify whether a particular structure of a language is "intended" or not to describe the architecture.

ADLs exist in two different forms: visual notations and text-based documents [9] [66]. This document focuses on text-based ADLs.

Describing the architecture can be considered a side objective in software production since it is not by essence needed to make an application work. Knowing the aim of the

language is then important to have it be useful. We can then think of an ADL as a means to help understand the system, giving priority to simplicity, or as a way to highlight and extract properties of the system, giving priority to strictness [66].

Although those general considerations tighten the scope of what falls under the denomination of ADL and give an idea about its purpose, there's *"little consensus in the research community on what an ADL is"* [66] and so we need to take a closer look at what precisely constitutes an ADL.

Technical definition of ADLs Instead of defining ADLs with a comprehensive definition, it can be defined through its fundamental constituents. The first two are the components of the system and their connectors to which has to be added architectural configurations i.e. abstract structures representing logical units of components and connectors with associated rules or properties of the system[9][66][47].

Components and connectors must define interfaces i.e. "*points of interaction with the external world*" as describing the connection between elements in a system can be seen as the principal object of architecture. Interfaces provide minimal information about the behavior of a component through the description of input and output, it is preferable for an ADL to be able to model further the behavior, ideally providing abstractions such as types that allow for compositionality and reusability[66].

But if we understand architecture as the *"fundamental concepts of properties of an entity in its environment and governing principles for the realization and evolution of this entity and its related life cycle processes*" [47], simply using components, connectors, and configurations to describe the state of a system at a point in time is not enough.

ADLs must also permit the creation and subsequent evolution of systems while guaranteeing the persistence of certain properties. In this perspective, at the component and connector level as well as at the system level with configurations, the language must provide ways to modify properties but also to express constraints that have to be respected during evolutions[66]. For reusability and easiness of maintenance, it is desirable that the language allows for typing and to make the system understandable from its sole description the behavior of components and connectors has to be expressible.

In short, an ADL describes the components of a system, the connectors between those components, and the general properties of the system. It must define explicitly the points of contact between the constituents of the system and allow evolution but also the expression of rules on the system and ensure that they are respected.

ADL Toolbox ADLs are usually a side document for the software, which make them an additional cost to create and maintain. Because of that their usage must be justified by benefits overweighting the costs. If the essential benefit they provide is the ability to share a common view of the architecture, they also come with an associated toolbox

that is specifically designed for the ADL to be able to render the service they were first designed for[66].

The most common aim for an ADL is to either be executable or provide a tool that produces executable code[21][9][66]. Doing so solves the problem of having to maintain the architectural description alongside the executable code since they are then the same thing or have the same source. For a language to be an ADL and thus provide a sufficient level of abstraction, it has to allow for different target implementations [9].

The second family of tools that can be provided with an ADL is meant to analyze the system. Those tools can be offline tools (static analysis), meaning that they apply to the architecture description, or online tools (dynamic analysis), meaning that they check the properties of the system as it's running. The offline tools are used to check if some properties are respected – such as those expressed in the components, connectors, and configurations descriptions – or simply if the described system is coherent[9]. That analysis can be extended to active specification where design options are only available when they do not conflict with expressed properties or system coherence. They can also advise the user based on existing type and suggestions about compositionality providing advice relative to common architectural structures such as design patterns[66][9]. Offline analytic tools can also help with code generation, verifying that rules on the system are not broken in the refinement process[66]. Online analysis can be used to check it properties are respected at runtime or to extract the architecture from the running system to either create or update the description files.

The last family of tools is used to represent the architecture. It generally consists in an abstraction of the system, either to present the architecture in different levels of complexity by displaying more or less information about the architectural elements or by using the indicated composition rules to present composites with their elements or as a whole. The representation tools also permit to produce different views of the system to fit the informational needs of different situations and stakeholders[66][9].

ADLs applied to Microservices Applications

Mapping ADL concepts to MS Considering that "a microservice architecture is a distributed application where all its modules are microservices"[25], mapping ADLs concepts to microservice applications is straightfoward enough.

The components are the microservices. The descriptions of the interfaces are usually a list of their endpoints but can also be seen at a higher level showing exposed ports. Typing is a possibility, particularly for infrastructural microservices that are common in multiple architectures. Default types are harder to think about for functional microservices but can be defined by users for a particular system or family of systems. Implementation instructions can be added by specifying a particular technology for a general microservice type. Constraints can be expressed at microservice level by indicating such requirement

as the type of protocol used for a connection (e.g. REST), connections with specific types of microservices (e.g. a database, a registry, a message broker) or being part of a larger structure (e.g. a registry being inserted in the "right" spot of a registry pattern).

The connectors are by essence more difficult to map to an object as they are an abstraction of the communication process happening between multiple entities. In a microservice context, besides the connected microservices, a key point is the protocol used to communicate. A type can also be relevant for connectors as simple as "one to one" or "one to many". Some abstract structures such as design patterns could then require some particular type of connectors. For instance a registry pattern would require a connector of the "one to many" form between the registry microservice and the others while a dedicated database pattern could require a "one to one" type of connector. Constraints on a connector could be a technical requirement such as a minimal bandwidth or a particular type of microservice to be connected to other. That last example somewhat blurs the distinction between connectors and architectural configurations as through semantics and constraint expression a connector can contain the logic of a particular pattern.

The configuration in a microservice context are identified groups of microservices linked with connectors in such a way that they represent a functional entity. They can be seen as the formal representation of design patterns and the constraints they induce. As such they can be of different natures. An instance can be the topology of a particular organisation (e.g. registry pattern, gateway pattern). Another architectural configuration describing the same object from a different point of view can address the workflow of such a topology, indicating the path of communication. A more specific architectural configurations can represent the workflow of a microservice system at a particular point in its life cycle (e.g. deployment). Architectural configurations are the elements of an ADL that bring the most in terms of understandability, compositionality as they reveal the hidden logic of an application. Because of that they are also they principal structure destined for analysing (online or offline) and representation.

Mitigation of Microservices Architecure issues using ADLs ADLs appear as a good way to mitigate microservices applications difficulties as they provide a global view to consider the difficulties emerging from the combination of those independent entities that microservices are.

The obvious challenge an ADL can help with is the design complexity inherent to microservices systems. Using architectural configurations, a representation tool can provide different levels of abstraction to help designing the system going from the higher levels toward the particularities of a specific microservice implementation. Once the application is online, the ADL is used to support the independence of teams as they share a common view of the global architecture, a more precise view of their vicinity and a detailed representation of their direct neighbors.

Another challenge of a different nature ADLs can assist with is the network load. Given enough informations relative to connectors and provided architectural configurations indicating the possible workflow in the application an offline analysis tool can check the conditions under which constraints set to represent a satisfying state of the application are met.

3.2 Studied Microservice ADLs: Presentation

This section presents and compare ten microservice ADLs: LEMMA, Silvera, MicroBuilder, MicroArt, Jolie, μTOSCA, Komponents, Compose and Kubernetes. They differ in multiple aspects such as their aims, their toolboxes composition or the point of view they adopt on microservice-based application. However they all fundamentally share the same underlying model, represented in Figure 4.1, which inhibit their ability to represent properly the architecture of application by having as first class citizens the microservices instead of the architectural structures. Because of that their application descriptions take the form of a list of microservices instead of bringing the acrchitecture on the forefront.

3.2.1 LEMMA

LEMMA [80] is a composition of four modelling languages giving four viewpoints on a microservice application. That decomposition is based on an extended reflexion on the different stakeholders and concerns relative to microservice architecture. The service viewpoint describes the microservices and their interfaces. The data structures viewpoint defines the persistent data and transient data structures. Those data structures can correspond respectively to those contained or linked to the microservices or to the data transmitted between microservices to exploit functionalities exposed through their interfaces. Those two viewpoints are essentially linked together and are the heart of the architectural description proposed by LEMMA. Through the description of data exchange and modification a limited description of business logic is provided. The deployment viewpoint describes the microservices defined in the service viewpoint with the properties relative to deployment. The last viewpoint is the technology viewpoint that allows to define specific implementations technology and specific attributes in need of specification for those technologies. They can them be applied to microservices in the service and deployment viewpoint to give a particular instantiation of the system. The possibility to define transient data structures communicated between microservices and having implementation technology as an additional layer are specific to LEMMA. The main tool provided by LEMMA is implementation generation in Java for the development part and with Docker Compose for the deployment part.
Listings 3.1, 3.2, 3.3 and 3.4 present a simplified description of an application consisting of three microservices, a gateway, a business microservice and a database. The business

```
context User{
  structure User<entity> {
    string username,
    string email,
    string password,
    immutable int id
  }

  structure UserService<service> {
    function User CreateUser(string username, string email,
       string password)
  }
}

context API {
  structure CreateUserRequest<valueObject> {
    string username,
    string email,
    string password
  }

  structure CreateUserResponse<valueObject> {
    int userId
  }
}
```

Listing 3.1: LEMMA Data Modeling Language simplified example

service `UserService` has its logic defined in the description by its data structure defined in Listing 3.1 and its interfaces defined in Listing 3.2. The database has its structure defined in Listing 3.3 (*lines 26-42*) and all three microservices have their deployment schema described in Listing 3.4.What is striking when looking at this architectural description that we presented briefly without entering its details is its complexity. Understanding the application requires a profound understanding of the language and even then, and for a very simple application, requires cross checking of multiple files. Even though LEMMA proposes a sophisticated syntax, it still presents an application as a list of elements, some being microservices presented through different points of view, some being related aspects such as technology implementation or data structure. In that regard its model is analogous to the one presented in Figure 4.1 offered in a variety of viewpoints. Because of that it does not provide structures dedicated to abstract the architectural structure of the application. Another limitation is the aim of the language, being code generation, which limits technology freedom to the implemented languages and connection to existing infrastructural microservices. Also the produced code has to be "refined" *i.e.* modified to be executable which implies a divergence with the original description leading with the application evolution for it to be incorrect.

```
functional microservice UserService {
  @endpoints (Spring::_protocols.rest: "/users";)
  interface Users {
    @required request
    @Spring::_aspects.Post
    create(
      sync in request : UserDomain::API.CreateUserRequest,
      sync out response : UserDomain::API.CreateUserResponse
    );
  }
}
```

Listing 3.2: LEMMA Service Modeling Language simplified example

```
technology Docker {
  deployment technologies {
    Docker {
      operation environment = "docker:25_0_3" default;

      service properties {
        string image = "openjdk:8u171-jre-alpine"<singleval>;
        string Dockerfile<mandatory, singleval>;
      }
    }
  }
}

technology Spring {
  protocols {
    sync rest data formats "application/json" default with format
        "application/json";
  }

  service aspects {
    aspect Post<singleval> for operations {
      selector(protocol = rest);
    }
  }
}

technology MySQL {
  protocols {
    sync connectorJ data formats "jdbc_enabled" default with
       format "jdbc_enabled";
  }

  infrastructure technologies {
    MySQL {
      operation environments = "mysql:5.7.13" default;
    }

    service properties {
      string rootPassword<mandatory, singleval>;
      string mysSqlUser<mandatory, singleval>;
      string mySqlPassword<mandatory, singleval>;
    }
  }
}
```

Listing 3.3: LEMMA Technology Modeling Language simplified example

```
@technology(Docker)
@technology(Spring)
container UserServiceContainer deployment technology Docker::_deployment.Docker deploys UserService {
  UserService {
    Dockerfile = '
      CMD java -jar user-service.jar
      '

    basic endpoints {
      Spring::_protocols.rest: "8080:8080";
    }
  }
}

container ApiGatewayContainer deployment technology Docker::_deployment.Docker {
  ApiGateway {
    Dockerfile = '
      CMD java -jar api-gateway.jar
      '

    basic endpoints {
      Spring::_protocols.rest: "80:80";
    }
  }
}

@technology(MySQL)
MySQL is MySQL::_infrastructure.MySQL used by nodes UserServiceContainer {
  default values {
    rootPassword = "rootpassword"
    mySqlUser = "mysqluser"
    mySqlPassword = "mysqlpassword"
  }

  endpoints {
    MySQL::_protocols.connectorJ: "3306:3306";
  }
}

ApiGatewayContainer uses UserServiceContainer via Spring::_protocols.rest;
```

Listing 3.4: LEMMA Operation Modeling Language simplified example

3.2.2 Silvera

Silvera [92] is an ADL dedicated to microservices. The description of an application is done through the description of its composing microservices. For each microservice information can be given about its communication style, its deployment and its interface which is a list of its endpoints. It is possible to indicate dependency between microservices, one expecting specific endpoints from the other. Some basic builtin typing for microservices is present that matches design patterns provided by the language (api-gateway, message broker and service registry). Silvera proposes a compiler based on code generators plugins converting the described architecture to executable code. By default the implementation is done in JAVA using the Spring boot framework and one selected technology for each builtin infrastructural microservice type. During compilation, the Silvera's compiler effectuates an analysis of the application and produces through six metrics a qualitative evaluation of the system and of its components.
Listing 3.5 shows a simplified description using the Silvera language of an application composed of four microservices. Three of them are explicitly described, `ServiceRegistry`, `EntryGateway` and `UserService`, and one, a database, is implied by the `UserService` definition (`lines 30-35`). Silvera implements types for infrastructural microservices (*lines 1 & 11*) and has keywords to specify in microservices description blocks linking to others (`lines 12, 17 & 23`). The architectural aspect is the main point of view of the Silvera language however, even if a series of patterns are implemented, they are not syntactically at the heart of the language but relegated in their expressions as online properties of microservices that are the first class citizens. Even if the explicit function of microservices is made immediately clear from their types and that we can infer patterns from them, the microservices they encompass have to be looked for in each defining block. That comes from the Silvera language having the underlying model represented in Figure 4.1 making its application description a list of microservices descriptions with some specific keywords for architectural properties. Another limitation of the Silvera language is that it aims a producing executable business code. The first consequence is the important reduction in technology freedom as the user is limited to the technology for which code generation is implemented. For instance, solely MongoDB is usable as a database. The second consequence comes from the necessary modifications, euphemistically called refinement, to be made on the produced code creating an irremediable, immediate and ever growing divergence with the original description, resulting in it being incorrect.

```
service_registry ServiceRegistry {
  tool=eureka
  client_mode=False
  deployment {
    port=9090
    url="http://registryexample.com"
    host=container
  }
}

api-gateway EntryGateway {
  service_registry=ServiceRegistry
  deployment {
    port=8080
    url="http://gatewayexample.com"
  }
  gateway-for{
    UserService as /users
  }
}

service UserService {
  service_registry=ServiceRegistry
  deployment{
    port=8080
    host=container
  }

  api {
    typedef User[
      @id int id
      @required str username
      @required str email
      @required str password
    ]

    @rest (method=POST)
    int createUser(str username, str email, str password)
  }
}
```

Listing 3.5: Silvera simplified example

3.2.3 MicroBuilder

MicroBuilder [97] is a combination of a microservice ADL (MicroDSL) and a code generator (MicroGenerator) used to describe and generate the code of REST microservice application. A particularity of MicroDSL is that it describes the datastructures associated with the microservices and differenciates between persistent data structures, that should be stored in databases, and transient data structures that exist only for the running time of the microservice and that are lost once execution stops. Microservices stored data, whether they are persistent or not are accessible through predefined REST methods (GET, POST, PUT, DELETE) upon which can be added custom REST methods whose implementation is described inline. While additional information relative to network connectivity can be added, MicroDSL is strongly oriented toward the data viewpoint.

From the application description Java code is generated using code generators.

Listing 3.6 presents a simplified application description written with the MicroBuilder language. The description defines explicitly two microservices `Gateway` and `UserService` and implies the instantiation of a service registry and the parallel existence of a database. There's no explicit link between `Gateway` and `UserService` but it is possible to infer it through the services and methods names and parameters. A dedicated database to `UserService` is assumed through the `EntityRessources` block (*lines 24-37*) defining the necessary parameters to interact with it. A service discovery pattern is indicated through the `AutoDiscovery: true` property (*lines 5 & 20*) implying the instantiation of a registry microservice at its center. This MicroBuilder example describes an application where the architectural structure is extractable through keywords destined to configure the microservices communications. However, this example also covers the full capacity of MicroBuilder for architectural expression, making it very limited in this regard considering the range of commonly used patterns as shown in Table A.2. The architecture is not made the central point of view which is a consequence of the language being structured by the model shown in Figure 4.1 making effectively the description an unorganized list of microservices defining blocks blending architectural properties and technical configurations. An other fundamental problem with MicroBuilder is that by having code generation as its ultimate production, it first reduces greatly technology freedom for the microservices it describes and as it needs generated code to be refined for it to be executable, it creates an unsolvable and ever growing divergence with the original description, making it rapidly useless.

3.2.4 MicroArt

MicroArt [37] takes the opposite approach compared to most of the other microservice ADL. It is firstly thought of as a tool of architecture reconstruction describing both the physical architecture and the logical architecture. To that effect an offline and online analysis of the application is done and its results are injected in the ADL structures. The tool then produces a dependancy graph of the application linking the endpoints of microservices to their callers. After that a manual refinement is done by the user to identify the service discovery microservice and remove its links to other microservices in order to get the logical dependancy graph of the application. The ADL contains typing of microservices, distinguishing between functional and a number of infrastructural microservices. Interfaces are explicited and a particular entity called "Link", linking a number of sources and target can be seen as a rudimentary connector. The language does not contain a textual concrete syntax.

```
MicroserviceArchitecture<'Microservice Example Application'>{
  Microservices:[
    Microservice <Gateway>{
      Port: 80
      AutoDiscovery: true
          RESTMethod: [
            RESTMethodType: POST
            RESTMethodReturnType: Integer
            RESTMethodParameters: [
              %Single String% username,
              %Single String% email,
              %Single String% password
            ]
            MethodURL: '/createuser'
            MethodBody : '/* JAVA code /*'
          ]
        ]
    Microservice <UserService>{
      Port: 8080
      AutoDiscovery: true
      Entity <User> {
        BaseURL: '/users/'
        IsPersistable: true
        EntityRessources: [
          %Single Integer% id [
            IsIdentifier: true],
          %Single String% Username,
          %List String% email,
          %Single String% password,
          RESTMethod: [
            RESTMethodType: POST
            RESTMethodReturnType: Integer
            RESTMethodParameters: [
              %Single String% username,
              %Single String% email,
              %Single String% password
            ]
            MethodURL: '/create'
            MethodBody : '/* JAVA code /*'
          ]
        ]
      }
    }
  ]
}
```

Listing 3.6: MicroBuilder simplified example

3.2.5 Jolie

Jolie [52] to the difference of other development oriented languages presented is not a description language but a programming language dedicated to services (though it can be compiled to Java). As such the implementation of the services' functionalities are written in line in a specific section of the service description. The specificity of Jolie is that the microservice application is thought as would be a monolithic application, with the help it provides (e.g. debugging), while keeping the independence of execution intact. For instance, as default behavior of a system written in Jolie, a type mismatch in a request will provoke an exception on the sender side. Another particular capacity of Jolie is that it allows to combine microservices to form new ones through synctactical structures that can be considered as architectural configurations. On the deployment side, Jolie microservices can be made into Docker images and deployed using Docker or Kubernetes or simply as independent processes. Jocker, a solution using Docker but making it transparent and allowing to use Jolie services contained in Docker containers as if they were not (i.e. as if they were directly deployed on the host) is in development.

Listing 3.7 presents a simplified application written with the Jolie language. The application is composed of two microservices, `UserService` and `Gateway`, and assumes the existence of an other microservice, a database, to which `UserService` connects to through the informations defined in `embedMySQLClient`. The connection between `Gateway` and `UserService` is detemined through the `inteface` keywords and the business logic contained in the `main` blocks contained in the `service` blocks defining the microservices. This example shows a description fitting the model represented in Figure 4.1, presenting a list of microservices without providing a construct expliciting the application architecture. It is to be noted that Jolie has an underlying model extending the model presented in Figure 4.1 as a service can contain other services. In that regard we can consider Jolie as having a concept matching the concept of configuration *i.e.* expressing the topology of a part of the application. To its credit, the Jolie language, aside from being an ADL, is also a complete programming language and thus does not need any post compilation refinement to produce a working application. Because of that, it is not prone to the problem of creating a irremediable divergeance between generated code and initial application description, making the latter rapidly outdated. However, the way a `service` object encompassing other `service` objects, is managed by Jolie at runtime is not by having the contained services run as independent processes but as combining them into a single process, thus loosing the fundamental aspect of the microservice style. Another aspect of Jolie appearing as contradictory to the microservice spirit is the will to use the language as the main, if not the only, technology used in the described microservice application thus removing the technology freedom. A corollary is that using Jolie as the main technology for writing microservice applications makes the user dependant to the language available implementation; for instance the standard and only fully developed

```
interface GatewayInterface {
  RequestResponse: registerUser(UserRequest)(UserResponse)
}

interface UserServiceInterface {
  RequestResponse: createUser(UserRequest)(UserResponse)
}

type UserRequest {
  .username: string,
  .email: string,
  .password: string
}

type UserResponse {
  .userId: int
}

embedMySQLClient(
  DB_HOST = "localhost",
  DB_PORT = 3306,
  DB_USERNAME = "mysqluser",
  DB_PASSWORD = "mysqlpassword",
  DB_DATABASE = "userdatabase"
) { .conn as db }

service UserService {
  inputPort UserPort {
    Location: "socket://localhost:9001"
    Interfaces: UserServiceInterface
  }

  main {
    createUser(request)(response) {
    // Logic to process request
    }
  }
}

service Gateway {
  inputPort HttpIn {
    Location: "socket://localhost:8000"
    Interfaces: GatewayInterface
  }

  outputPort ToUserService {
    Location: "socket://localhost:9001"
    Interfaces: UserServiceInterface
  }

  main {
    registerUser(request)(response) {
      createUser@ToUserService(request)(response)
    }
  }
}
```

Listing 3.7: Jolie simplified example

communication means between microservice is direct communication which by nature does not provide fault-tolerance which is another one of the main advantages of the microservice architectural style.

3.2.6 JHipster

Jhipster [50] is an application generator that uses a microservice ADL named JDL [51] (JHipster Domain Language) to describe the technology stack and topology of a Microservice application. The application architecture is not flexible as it always consist in a gateway pattern under which functional microservices are connected through a registry pattern. Modifications to the architecture have to be done once the executable files have been generated as JHipster can be considered a boostrapper providing boilerplate code of interconnected technologies. Beside describing the technology stack, the focus of the language is the definition of the data structures owned by the microservices and their relations. The language only proposes the "database per service" pattern (it is not possible to express the sharing of databases). The language contains only one distinction of microservice type, between functional microservice and gateway. The registry is not explicitly mentioned but created at compile time. A text editor checking the correctness of the syntax is proposed online as well as a plugin for the IntelliJ IDE.

Listing 3.8 presents a simplified application written with the JDL language. The application is composed of four microservices, with only two represented in the description. `gateway` and `userService` are described explicitly in the application, the relation between the two being indicated as an inline property by specifying the `applicationType` attribute of `gateway` as a gateway for all other business microservices in the application thus indicating an API gateway pattern. Two other infrastructural microservices are present in the application. The first one being a registry indicated through `serviceDiscoveryType` attributes (*lines 6 & 18*) indicating a service registry pattern by specifying the technology for it as an `eureka` microservice. The second one is a database declared inline with the `databaseType` and `prodDatabaseType` (*lines 19-20*) indicating a dedicated database to the `userService` microservice. The data structure of this database is specified in the `entity` block defining a User table linked inline to the `userService` microservice (*line 24*).

This example of the JDL language shows a case where the structure of the application is described explicitly with dedicated keywords. However, this example covers the full extent of the ability of the JDL language for pattern expression, and no other patterns listed in Table A.2 could be explicitly expressed. Arguably this limitation could be overcome by extending the language. However, this argument only stands for widely used and identified patterns and could not be applied to niche application. More fundamentally the architecture is not put at the front of the discussion, and implies to dig in microservices descriptions to understand their roles and even their existence as most infrastructural

microservices are not first class citizen but expressed as inline properties. Those two difficulties are intrinsically linked to JDL underlying model, represented in Figure 4.1, not providing any concept to represent the application architectural structures.

```
application {
  config {
    baseName gateway,
    applicationType gateway,
    packageName com.example.gateway,
    serviceDiscoveryType eureka,
    buildTool gradle,
    authenticationType jwt,
    serverPort 8080,
  }
}

application {
  config {
    baseName userService,
    applicationType microservice,
    packageName com.example.userservice,
    serviceDiscoveryType eureka,
    databaseType sql,
    prodDatabaseType mysql,
    buildTool gradle,
    serverPort 8081,
  }
  entities User
}

entity User {
  id Long required,
  username String required,
  email String required,
  password String required
}
```

Listing 3.8: JHipster Domain Language simplified example

3.2.7 μTOSCA

μTOSCA [89] is an architecture description language describing microservice applications. It is an extension of the TOSCA specification [74], an OASIS standard aiming at describing cloud applications, while being technology agnostic but processable so that the deployment could be automated. The concrete syntax proposed for both TOSCA and μTOSCA is in YAML. TOSCA already supports typing and inheritance as well as groups to define policies on multiple nodes. Connections between components, expressed with the `relationship` keyword, are not first class citizen in architecture description although they posses their own independent typing system. μTOSCA does not modify the metamodel of the TOSCA specification but specifies its functionalities for the microservice architecting style. It then provides a number of microservice types, e.g., DataStore, relationship, e.g., InteractsWith and a particular group named `edge` that

gathers the identifiers of the microservices accessible from outside of the application. Most importantly μTOSCA retains the advantage of the powerful extensibility of the original TOSCA specification allowing the user to define any new type of service, relationship or group. The more consequential addition to TOSCA is μTOSCA toolchain whose aim is to extract and correct the architecture of microservice applications. The first tool, μMiner [72] [71] takes the Kubernetes manifests of an application, applies static then dynamic analysis to extract the architecture and express it in the μTOSCA language. The μTOSCA files can then be passed to μFreshener [5] [88] which provides a graphical representation of the application, identifies anti-patterns and proposes corrections.

Listing 3.9 presents a simplified microservice-based application written with the μTOSCA language in YAML format. The application is composed of three microservices, `gateway` is the point of entry in the application, `userService` is the business service containing the application logic and `mysql` is its dedicated database. To describe the application in a more precise way than what the builtin functionalities allow for, we define two new subtypes. The first one `DataBasePerService` (*lines 1-4*) is a new type of group and is meant to represent the pattern of the same name, using two existing μTOSCA types, a `Service` and a `DataStore`. The second `ServiceWithPort` (*lines 5-11*) is a new type of service, meant to represent a service with a port exposed to other services in the applications. The `topology_template` keyword (*line 12*) indicated the actual description of the application. It is composed of two parts. The first one is mandatory and lists exhaustively the microservices in the application, using their identifier as key for the properties defining them (*lines 14, 24 and 36*). Their defined properties possess at a minimum their type and optionally requirements such as their relations with other microservices (*lines 17-21 and 29-33*) or the underlying deployment architecture. Those relationships are typed and allow to represent certains patterns. Here the `gateway` microservice uses the pattern `dynamic_discovery` (*line 21*) to automatically retrieve the address of `userService` which allows for its horizontal scaling. On the contrary `userService` explicitly indicates that it does not use the discovery pattern in its relation with the service `mysql` (*line 33*) thus implying that all the possible replicas of `userService` connect to the same `mysql` instance. The second part of the architecture description, `groups` (*lines 40-46*) is optional and describes concerns shared by multiple services. Its purpose is mainly, and here entirely, for documentation purposes and aims at highlighting policies applied in the application. In this example, `Edge` which is the only group with an architectural reach provided by μTOSCA is used to gather the services open to extenal communication (*lines 41-43*). The other group that we previously defined `DatabasePerService` is then used (*lines 44-46*) to make the pattern linking `mysql` and `userService` explicit.

```
group_types:
  micro.groups.DataBasePerService:
    derived_from: micro.groups.Root
    members: [ micro.nodes.Service, micro.nodes.DataStore ]
node_types:
  app.nodes.ServiceWithPort:
    derived_from: micro.nodes.Service
    properties:
      container_port:
        type: integer
        required: true
topology_template:
  node_templates:
    gateway:
      type: micro.nodes.MessageRouter
      requirements:
        - interaction:
            node: userService
            relationship: micro.relationships.InteractsWith
            properties:
              dynamic_discovery: true
      artifacts:
        container_image: gateway
    userService:
      type: app.nodes.ServiceWithPort
      properties:
        container_port: 8081
      requirements:
        - interaction:
            node: mysql
            relationship: micro.relationships.InteractsWith
            properties:
              dynamic_discovery: false
      artifacts:
        container_image: userService
    mysql:
      type: micro.nodes.DataStore
      artifacts:
        container_image: mysql:5.7
  groups:
    edge:
      type: micro.groups.Edge
      members: [ gateway ]
    user_data_domain:
      type: micro.groups.DataBasePerService
      members: [ userService, mysql ]
```

Listing 3.9: microTosca simplified example

3.2.8 Komponents

Komponents [29] [48] is a modeling language aiming at describing microservice-based applications. It focuses on deployment and operation properties, is technology agnostic and has a YAML concrete syntax. The components of the language are of three types, `basic`, `composite` and `deployment`. The `deployment` type is used to represent applications, distinguishing executable elements, i.e., `basic` and `composite` constructions, which are defined in the `import` attribute and referred to in the `subcomponents` attribute, from

the connections between them which are explicitly specified in the `connectors` attribute. The `basic` type is used to define individual microservices while the `composite` type is used to wrap multiple `basic` objects in a reusable structure. The `composite` type cannot be extended directly after being imported but can be wrapped inside new `composite` objects to which properties can be added. However `basic` objects have to be defined in a single place thus making effectively Komponents applications descriptions lists of microservices with higher degree structures meant to represent architectural constructions. The entrypoints of the application are listed in the `entrypoint` attribute in order to make them easily identifiable. The Komponents language is accompanied by a graphical tool to model applications in a web interface and produce from the schema Komponents files. Another tool allows to deploy applications on both Docker and Kuberbetes.

Listing 3.10 and Listing 3.11 represents a simple application compose of a gateway leading to a business microservice with a dedicated database, it describes an object of type `deployment` (*line 1*). A single entrypoint in the application is determined (*lines 3-6*). The applications description is done under the `model` attribute, which refers to an object of type `composite` composed of four parts: the `variable` section (*lines 9-15*) provides the values of the global variables in the application description, the `import` section (*lines 16-67*) where the components are defined, the `subcomponents` section (*lines 68-71*) where the components deployed are listed and the `connectors` section (*lines 72-93*) that lists connectors between the deployed components. The `import` section can hold either `basic` or `composite` objects definitions. Here the three defined objects: `Gateway` (*lines 17-28*), `UserService` (*lines 29-39*) and `Database` (*47-67*). Ideally we would have wanted to regroup services in two `composite` objects representing the Api gateway pattern and the database per service pattern. However, Komponents does not allow for `Basic` objects to be part of multiple composite and `UserService` is part of both patterns. We could have chosen to make it only part of one of the two but then it would have overshadowed the other. Looking at `UserService` description, we find the expected properties for the definition of a microservice from a deployment and operation point of view. The identifier is given through the `name` attribute (*line 30*), the image for the business code with the `source` attribute (*line 33*) and the environment variable defined in the `variable` attribute (*lines 35-39*). The operation scaling range is specifiable with the `cardinality` (*line 34*) attribute. The endpoints are described explicitly under the `endpoints` attribute (*lines 40-46*), indicating that `UserService` accepts connections on the `8081` port and specifically connects to the port of the database whose value is set through a variable shared across to the whole application description. The `subcomponents` attribute of the `model` section lists the deployed components and associates them with new identifiers that are used in the `connectors` we note that having decided that `UserService` is horizontally scalable, the connection to it coming from `Gateway` is done through a LoadBalancer while since there's a single `Database` instance the connection to it coming from `UserService` is done through a simple link. Finally, the `endpoints` section (*lines 89-93*) instructs the

```
type: deployment
name: user-app
entrypoints:
  gateway:
    protocol: tcp:8080
    mapping: gateway
model:
  type: composite
  variables:
    DB_HOST: mysql
    DB_PORT: 3306
    DB_USER: mysqluser
    DB_PASSWORD: mysqlpassword
    MYSQL_ROOT_PASSWORD: root
    MYSQL_DATABASE: userServiceDb
  imports:
    Gateway:
      name: Gateway
      type: basic
      runtime: docker
      source: gateway
      endpoints:
        in:
          type: in
          protocol: tcp:8080
        user:
          type: out
          protocol: tcp:8081
    UserService:
      name: UserService
      type: basic
      runtime: docker
      source: userService
      cardinality: "[2:10]"
      variables:
        DB_HOST: "{{ DB_HOST }}"
        DB_PORT: "{{ DB_PORT }}"
        DB_USER: "{{ DB_USER }}"
        DB_PASSWORD: "{{ DB_PASSWORD }}"
      endpoints:
        in:
          type: in
          protocol: tcp:8081
        db:
          type: out
          protocol: "tcp:{{ DB_PORT }}"
    Database:
      name: MySQL
      type: basic
      runtime: docker
      source: mysql:5.7
      cardinality: "[1:1]"
      variables:
        MYSQL_ROOT_PASSWORD: "{{ MYSQL_ROOT_PASSWORD }}"
        MYSQL_DATABASE: "{{ MYSQL_DATABASE }}"
        MYSQL_USER: "{{ DB_USER }}"
        MYSQL_PASSWORD: "{{ DB_PASSWORD }}"
      endpoints:
        in:
          type: in
          protocol: "tcp:{{ DB_PORT }}"
      volumes:
        initdb:
          path: /docker-entrypoint-initdb.d
          scope: local
          durability: permanent
         url: ./initdb
```

Listing 3.10: Komponents simplified example (Part 1)

```
  subcomponents:
    gw: Gateway
    us: UserService
    db: Database
  connectors:
    lb-gw:
      type: LoadBalancer
      outputs:
        - gw.in
    lk-gw-us:
      type: LoadBalancer
      inputs:
        - gw.user
      outputs:
        - us.in
    lk-us-db:
      type: Link
      inputs:
        - us.db
      outputs:
        - db.in
  endpoints:
    gateway:
      type: in
      protocol: tcp:8080
      mapping: lb-gw
```

Listing 3.11: Komponents simplified example (part 2)

underlying execution platform where the possible route for traffic that has entered the application.

3.2.9 Compose

Compose [34] is a language used to describe and configure "multi-container applications". In that regard it is devoted to a particular instantiation of microservices. The main implementation of the language is its Docker implementation[75]. Being a deployment language, it allows to define some properties of the system relative to the environment in which the containers are running such as allocated resources, network connectivity or data storage and properties relative to the environment inside the container such as environment variables or command to be executed at system startup. Through network connectivity, Compose describes interfaces of microservices in the form of ports exposed to the other microservices and of the system with ports exposed to the outside world. If Compose does not have a notion of connector, it is possible to express dependencies between services at deployment time. It is to be noted that while its first purpose is not architecture description, unlike most ADLs it allows to express properties on groups of microservices such as belonging to the same network or shared storage. There is no typing in Compose.

Listing 3.12 presents a simplified microservice-based application written with the Compose language in Yaml format. The application is composed of three microservices,

```
version: '3.8'
services:
  userService:
    image: userService
    environment:
      - DB_HOST="mysql"
      - DB_PORT=3306
      - DB_USER="mysqluser"
      - DB__PASSWORD="mysqlpassword"
    depends_on:
      - mysql
    expose:
      - 8081

  gateway:
    image: gateway
    depends_on:
      - userService
    ports:
      - "8080:8080"

  mysql:
    image: mysql:5.7
    environment:
      - MYSQL_ROOT_PASSWORD: root
      - MYSQL_DATABASE: userServiceDb
      - MYSQL_USER="mysqluser"
      - MYSQL_PASSWORD="mysqlpassword"
    ports:
      - "3306:3306"
  volumes:
      - ./initdb:/docker-entrypoint-initdb.d
```

Listing 3.12: Compose simplified example

`gateway` is the point of entry in the application, `userService` is the business service containing the application logic and `mysql` is its dedicated database. The two microservice architecture patterns structuring the application are not made explicit by the language and can only be deduced by an analysis of the properties. The dependencies indicates a relation between the microservices. `userService` appears to be at the center of the application, with its connection to `gateway`, forming and API gateway pattern, being indicated by the `depends_on` property at *lines 17-18* and its connection to `mysql`, forming a database per service pattern, indicated also by the `depends_on` property at `lines 10-11` and by the properties related to the `environment` attribute at `lines 5-9` indicating informations necessary to connect to `mysql`. Even though we have shown that we can understand the application structure through analysis, none of the information we used are a necessity in the description as they could be set thought other means. In addition in a more realistic and complicated application extracting the architecture would pose a greater challenge and a proportional increase in possibility of errors. That difficulty is intrinsically linked to Compose underlying model, represented in Figure 4.1, not providing any concept to represent the application architectural structures. However, because Compose defines more the deployment and operational properties surrounding microservices, it does not infringes on the implementation and thus leaves intact technology freedom. Because of that and of its declarative nature it is a prime candidate for being the compilation target of a microservice ADL providing the appropriate abstractions to describe applications architecture.

3.2.10 Kubernetes

Kubernetes [43] is a system used to configure, deploy and manage containers. Interaction with the system is done through files named *manifests* declaring a state of the application. Kubernetes then either deploy or modify the system so that it matches the description. Kubernetes shares a good part of configurations options contained in Compose but adds its own structures on top of them. Kubernetes' more abstract and particular concepts are sometimes substituted to Compose concepts that are relatively close to the real network entities. For instance, opening passage from the outside to a particular service in Compose is done through the "Port" keyword in the service description. In Kubernetes a particular object named *service* will act as an intermediary. Kubernetes has the ability to define interior and exterior interfaces with ports but can also manage the routing of endpoints through a specific object called *Ingress*. Requests going though the Ingress resource are routed to undetermined instances of a particular microservice. Even thought the description of an application through manifests is done using what can be considered as arbitrary concepts, it describes a logical abstraction of the application that is understandable for the user. Because of that it provides a description that is arguably

more architectural than some obtained using other ADLs and thus qualify as one.
Listings 3.13, 3.14 and 3.15 present a simplified microservice-based application written with the Kubernetes language in Yaml format. The application is composed of three Kubernetes resources, two of which, `user-service` and `mysql`, existing as processes at runtime, and one as an Ingress Kubernetes resource acting logically as a gateway but being effectively a configuration of the Kubernetes software. Kubernetes manifests are strongly linked to the underlying software of the same name acting as an hardware abstraction. Most of the keywords are specific to the technology and are not representing the abstracted architecture of the application but configure the management of the microservices at runtime. Nonetheless the architecture is in this example extractable through analysis from the microservices properties with for instance the connection of `user-service` to `mysql` being expressed in the defined environment variables of the former (*lines 28-36* of Listing 3.13). The complete understanding of the architecture from a Kubernetes application description is then a tedious and error prone process, even more so because Kubernetes is focused on managing large applications. That comes from Kubernetes having an underlying model largely analogous to the one presented in Figure 4.1 making an application description a list of resources definitions. We note that in some specific cases and for technical reasons it is possible to regroup microservices in encompassing structures (`Deployment`) matching the topology of some microservice architectural patterns such as sidecar. However it is a specific use that does not represent the common usage where microservices are individually contained in Kubernetes specific structures (`Service` and `Deployment`). One interesting aspect of Kubernetes is that being an executable language its descriptions are exactly mapped to the executional reality and that being a deployment and operational language it does not infringe on technological freedom in the business implementation. While it misses proper structures to represent microservice-based architecture, its declarative nature make it a prime candidate for compilation target of a microservice ADLs offering the appropriate abstractions to do so.

```
apiVersion: v1
kind: Service
metadata:
  name: user-service
spec:
  ports:
  - port: 8081
    targetPort: 8081
  selector:
    app: user-service
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: user-service
spec:
  selector:
    matchLabels:
      app: user-service
  template:
    metadata:
      labels:
        app: user-service
    spec:
      containers:
      - name: user-service
        image: userService
        env:
        - name: DB_HOST
          value: "mysql"
        - name: DB_PORT
          value: "3306"
        - name: DB_USERNAME
          value: "mysqluser"
        - name: DB_PASSWORD
          value: "mysqlpassword"
        ports:
        - containerPort: 8081
#
#
#
```

Listing 3.13: Kubernetes user service simplified example

```
apiVersion: v1
kind: Service
metadata:
  name: mysql
spec:
  ports:
  - port: 3306
  selector:
    app: mysql
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mysql
spec:
  selector:
    matchLabels:
      app: mysql
  strategy:
    type: Recreate
  template:
    metadata:
      labels:
        app: mysql
    spec:
      containers:
      - image: mysql:5.7
        name: mysql
        env:
        - name:
           MYSQL_ROOT_PASSWORD
          value: "root"
        - name: MYSQL_DATABASE
          value: "userServiceDb"
        - name: MYSQL_USER
          value: "mysqluser"
        - name: MYSQL_PASSWORD
          value: "mysqlpassword"
        ports:
        - containerPort: 3306
          name: mysql
```

Listing 3.14: Kubernetes database service simplified example

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: user-service-ingress
spec:
  rules:
  - http:
      paths:
      - path: /createUser
        pathType: Prefix
        backend:
          service:
            name: user-service
            port:
              number: 8081
```

Listing 3.15: Kubernetes simplified Ingress resource

3.3 Classification Criteria

The analysis of the different microservice ADLs has to be split in two categories: the properties of the language itself and the tools furnished with it.

The criteria used to compare the languages are the following:

- **Components**:
  - Interface: The ability to express the points of connections to a component. It can either be a point of connection to the other components or a point of connection to the outside world. In the case of microservices it can be either ports or endpoints.
  - Types: The ability for a language to classify microservices. It can be in the form of builtin types, furnished by the language, or the abitilty given to the user to create custom types. The custom types are particularly relevant for functional microservices as they are specific to each microservice application and thus cannot be provided beforehand. The infrastructural type can be subtyped with common infrastructural microservices such as registry, gateway, message broker...
  - Semantics: The ability for a language to describe the behavior of a language from the outside. It can consist in the description of states indicating the functionalities provided by a microservice depending on previous interactions with it. It can also describe the consequences of an interaction with a microservice on the rest of the system (i.e. analysing, from an initial call on a microservice, the induced calls on its neighbors).
  - Constraints: The ability for a language to express constraints that must be respected. Those constraints can be relative to different domains such as connectivity, response time or implementation. A simple example would be

that a particular microservice should not be directly connected to the outside world or that it should be connected to a dedicated database.

- Implementation: Guidelines relative to the implementation of a microservice. For instance indicating a required or forbidden technology. It can also be the application of a layer or modification of a property indicating a particular technology. In some cases executable instructions can be specified in the description of a microservice. That category is obviously filled for executable languages.
- Design patterns: The ability for a microservice to express that it belongs to a particular design pattern and its role in it. For instance in the "database per service" pattern, the functional microservice can indicate inline which microservice is its dedicated database and inversely.
- Datastructures: The ability to describe the datastructures existing in the application. The most immediate datastructures to be described are the ones associated with functional microservices whose fields are directly involved in the functionalities given by the microservice. Other datastructures such as those transmitted to query the microservices can be specified.

- **Connector**: The presence in a language of a concept of connector which allows to describe the link between components. Connectors are first class entities of an ADL and as such have to be distinguished from the constituents of the interface of components with which it can share some informations. They can be classified using most of the categories used for components.

- **Configuration**: the ability to express architectural structures composed of components and connectors. The most natural structures that can be expressed are graph of components linked with connectors. Doing so a composite is formed providing an abstraction that can be considered and used as a component (Compositionality). The reusability provided by architectural configurations is either the duplication of entirely described parts of applications or patterns where only some fields need filling. Through this approach design patterns can be expressed and abstracted from particular applications, slots for microservices of different natures having different roles can be expected and rules concerning the inner workings of the architectural configuration can be expressed (Constraints). The simple listing of components and connectors admittedly describes precisely the system but does not make sense of it while on the other hand architectural configurations express by essence a superior level of abstraction that highlights the logic of the application. For that reason they can be considered as the concept that makes a language "architectural". Simplicity of syntax, relevance and broadness of application are fundamental for architectural configurations as they mostly determine understandability. However they are difficult to evaluate.

- Scalability: The possibility to express rules relative to how the system should scale vertically (i.e. adjusting the amount of ressources allocated to deployed elements) and horizontally (i.e. adjusting the number of microservices instances deployed) depending on system usage. That feature can exist for components, connectors and configurations.

- Decomposition: What is the main decomposition of the language, what are the real-world objects that are represented by components and, with a lesser degree of importance, connectors.

- Focus: Besides the main decomposition of the language, what is the aspect of microservices applications the languages focuses on.

The criteria used to classify the toolboxes are the following:

- Implementation: The possibility to execute the described system from its description. This tool allows to bridge the gap of having on one side the code of the application and on the other its description. Doing so it removes the necessity of keeping the architectural up to date which if not done renders the system description not only useless but potentially harmful. The implementation tool, can be a way of directly executing the system such an interpreter or a way to compile the description to an existing executable target language. In the first case, given that the executional implications of the language is well understood, the execution matches perfectly the description. The downside, apart from the considerable effort needed to produce such a solution, is that, it limits the technology freedom given to developers that is one of the main appeal of microservices architecture. In the second case, the ADL can be technology agnostic and different code generators can possibly be applied to different parts of the system. The maintenance of the tool is then a concern as the generators have to be kept up to date with the target languages. As the architecture language is often much more high level than the target language, the transformation phase has to make considerable choices as to how to realize the description. It is sometimes possible to express guidelines relative to how the system must be implemented in the architectural description of the system. It can go as far as introducing some target language code directly in the architectural description, be a specification phase happening between initial design and generation or a refinement phase happening after generation and modifying the produced code to match the expectations of the developer.

- Analysis: The analysis of the system can either be done offline, considering only the documents describing the system, and online, observing the system as it is running. The offline approach tends to check for more theoretical, general, less precise properties of the system since what is actually provable is limited due to the expression capacity of the language, and the specifics of code implementation

and infrastructure runtime. Online analysis however can verify that any property is respected, although what it gains in broadness it loses in depth as online analysis has only the ability to ensure that a property has been respected until now but cannot predict with certainty what will happen in the future. For those two, very different but not antagonistic, approaches it is possible to extract specific metrics, measuring a particular aspect of the system through a chosen quantification, or verify given constraints expressed as statements with the concepts furnished by the language or implicitly implied by choices made in the design phase. For instance a constraint expressed explicitly could be forbidding the entrance of outside communication except though one microservice and that exact constraint would be implicitly checked when a "gateway pattern" has been used.

- Active Specification: Providing a tool that gives architectural advices based on the current system descriptions.
- IDE Integration: Providing a dedicated integrated development (IDE) or modules for existing IDE that provide help for using the language such as syntax highlighting, syntax verification, snippets...
- Architecture Reconstruction: Providing a tool that can analyse a running application and produce a description of it in the related ADL.
- Visualization: A visualization tool produces a visual representation based on the description of the system. The first purpose of this tool is to help understandability of the application. It is particularly useful for text ADLs; it allows to have an idea of the system architecture with a quick glance when reading, connecting and abstracting from the text description can be quite difficult and even error prone for someone not accustomed with a particular application. Even for someone actively working on an application, it helps to have a simpler representation of the system available at all time. A great feature that can be added to that tool is the possibility to represent the system in different levels of abstractions. For instance a level of representation could be the individual microservices, a higher level could be the different communicational patterns and at an even higher connection level the application can be the component represented in its ecosystem. Those different representations, in addition to help developers keeping in mind "the big picture", can fit different kinds of stakeholder. Stakeholders interested only in the business side may not be interested in getting into the microservice decomposition but only in how the application is integrated in a larger ecosystem.

3.4 Studied Microservice ADLS: Languages

To provide a better understanding of the different properties of the solutions we have identified, we have decided analyse first the languages in their descriptive features then to compare the toolboxes provided with each of them. The compared properties are those presented in Section 3.3 To the exception of "Decomposition" 3.3 and "Focus" 3.3, all the criteria are assigned a value of true, indicating the presence of the feature, or false, indicating the absence of the feature. When a feature is part of language or a tool furnished in the toolbox, it is represented with a green cell containing the letter "Y", absent features and tools are represented by a red cell containing the letter "N". When it is not clear whether a feature should be considered part of a language or toolbox, the cell is filled in orange and marked with the "~" symbol.

To ease readability, we decomposed the languages classification table in four different tables: component description features table, connectors and configuration table, a table gathering other aspects of microservices ADLs and a table indicating the base concept of the language and what it focuses on. That last table contains classification criteria that either would not fit under one of the main sections of the other tables or on the contrary would fit in multiple ones but for which it was more relevant to assert their presence or absence in the language regardless of where.

3.4.1 Components

For all selected languages, the components are the microservices so we use both terms interchangeably. To be more precise Compose and Kubernetes components are respectively containers and workoads that refer to the execution environment which encompasses more that just the microservice as a program. However those entities can arguably be considered as a representation of the running microservice.

Table 3.1 represents the presence of different features in the different solutions:

Language	Interface	Types	Semantics	Constraints	Implementation
LEMMA	Y	∼	N	N	Y
Silvera	Y	Y	N	N	N
MicroArt	Y	N	N	N	N
Jolie	Y	N	Y	N	Y
MBuilder	Y	N	N	N	Y
JHipster	N	Y	N	N	Y
μTOSCA	Y	Y	N	Y	Y
Komponents	Y	N	N	N	Y
Compose	Y	N	N	N	Y
K8s	Y	N	N	N	Y

Table 3.1: Comparison of components (microservices) description in Microservices ADLs.

The "Interface" column determines if the ADLs explicitly define the interfaces of their components. We immediately notice that it is a shared feature for all the languages but one. Only JHipster does not indicate precise connection points. Connection to and through the database is either generated automatically at compile time using a default configuration or specified through configuration files. The communication between microservices inside the system is generated automatically based on the databases and their relationships that are explicitly indicated in the system description.
The others ADLs allow to explicitly define the microservices' interfaces. All of them allow to determine the exposed port for the microservice which we have considered a sufficient condition to check the interface description box as it is the point of entry into the application while endpoints indicate which functionality is solicited. Despite that indicating the endpoints gives a much more precise vision of what the microservice is through their names, that usually fit the functionalities, and the possibility to establish more sophisticated dependency links.
Apart from JHipster and Compose, all the ADLs allow to specify the endpoints. For LEMMA, Silvera, Jolie and MicroBuilder, an endpoint signature corresponding to a functionality provided by the microservice is writable inline in the microservice description that indicates the expected parameter and the returned datatype. Component wise Kubernetes only allows to express the port number of the exposed ports of the pod and associated service but a structure called Ingress allows to list the endpoints and their destinations.
Components are the base material of any ADL and interface description being a prerequisite to describe the architecture, it is fairly natural that we observed components' interfaces as an important focus of almost every ADL.

The "Types" column refers to the presence of types of microservices. As discussed in the Criteria Section (3.3), the main distinction in microservices types is made between

functional and infrastructural microservices. It is possible to expand on the latter with subtypes matching infrastructural microservices commonly found in applications.
As shown by the table it is an uncommon feature of ADLs to provide typing for microservices which can be surprising as it is a distinction that is commonly made when describing microservices applications and that multiple technologies dedicated to a particular aspect of microservice infrastructure are available and widely used.
The reasons for the lack of this important feature is that it doesn't align with most ADLs purpose.
LEMMA, Jolie and MicroBuilder are focused on describing the functional microservices, having infrastructural microservices being either implicit and added at compile (MicroBuilder) or added as a supplementary layer (LEMMA). Some of those languages (LEMMA, Jolie) produce deployment files either in Docker Compose or Kubernetes format so they can rely on the fundamental infrastructural microservices those technologies provide. About functional microservices, they are by essence hard to generalise as they are developed for a specific purpose. It is then reasonable to assume that adding types in those languages seemed irrelevant, useless or even harmful to their creators.
In LEMMA, infrastructural microservices can be defined in the "technology viewpoint" and then integrated in the system but only in the "operation viewpoint" limiting their architectural presence in the system description. While it is a de facto type distinction, it also sets aside those microservices considering they are somewhat secondary in the architecture relegating them to the technology aspect of things. Because of that and the absence of typing on LEMMA main focus (i.e. functional microservices), we considered the language to be in an intermediate situation relatively to typing.
Compose and Kubernetes are focused on the configuration of the environment. Part of what would be considered infrastructural is already considered as taken care off in the settings they describe (discovery, routing) and for the particular infrastructural microservices used by an application, there is no obvious reason from an operation standpoint to consider them differently than any other microservice. Kubernetes possesses the Ingress resource that acts as a gateway but is more akin to a specific microservice than a type.
The languages having a concept of type are Silvera and JHipster. JHipster distinguishes only one type from functional microservices which is the "gateway" type. That distinction is driven by the aim of JHipster as a whole that is to produce an advanced prototype of a web application where the user can define the technology stack associated with functional microservices. The application always has a gateway that produces a different code than the other microservices. Silvera also contains a gateway type in which, to the difference of JHipster, the path routing can be specified. Silvera also has a registry type acting as a wrapper for a specific register technology that is specified in a field.
MicroArt has no concrete syntax but a metamodel used to make a model of the application through the online analysis of the application. Its metamodel contains a notion of type

which is effectively unused so we tend to consider that it does not provide typing for microservices.
In summary, typing is almost non existent in the ADLs we have investigated, the reasons we have identified is the difficulty of determining useful and relevant types for functional microservices, the implicit presence of infrastructural microservices or the absence of a need to identify them as specific microservices.

The "Semantics" column determines if the language allows for a description of the behavior of the component from the outside (see discussion in the "Classification Criteria" Section 3.3). To the exception of Jolie, it is a missing feature of all ADLs we have considered. The closest feature to semantics description that some languages provide (LEMMA, Silvera, MicroBuilder) is the indication of the data type transmitted when responding to a specific functionality call. However we regard that as insufficient to affirm that those languages provide semantics descriptions as it belongs to endpoints interface description and that it does not give information to the induced state of the microservice *i.e.* the change it implies on offered functionalities.
Jolie on the other hand has a specific, simple and appropriate syntax allowing to describe the impact of interactions with the microservice on its subsequent provided functionalities[39]. Using the keywords "provide" and "until" it describes the behavior of the microservice as a machine state.
Compose provides a dependency configuration feature that indicates the order in which the system must be deployed and through online analysis MicroArt is able to express dependencies between endpoints. Those two approaches cover some elementary parts of what describing the semantics in the system in the sense of describing the communicational impact would be. However, we consider that to properly express that acceptation of semantics it is necessary to establish the link between incoming and outgoing communications. Because of that we consider that no language describes the communicational behavior of microservices. In summary, semantics description is a largely unexplored aspect of microservice applications architecture description. We presume that it comes from the difficulty of establishing an appropriate model and syntax for it but also because most of the ADLs we have looked at are oriented toward code generation and thus leave that aspect of microservice applications to their concrete implementation. We consider that as a deficiency as semantics are a considerable part of what makes a language architectural. Moreover, in the context of code generation, semantics description could make the generated much closer to the architect intent thus reducing the manual refinement process. Jolie being a dedicated microservice programming language does not have that discrepancy and pertinently cover that aspect.

The "Constraints" field determines if it is possible to express statements that microservices must respect. Coupled with the appropriate tool, it allows the architect to be sure

that its component description does not break those general rules. It is also a way to ensure that it won't happen in future modifications of the system and even if there is no verification it has the merit of explicitly stating the imperatives relative to a particular component.
We consider that a language would be able to describe constraints if it can do so in a direct manner and not as a consequence of another description. For instance, LEMMA allows to specify the protocol of an endpoint. We consider this as a description of the endpoint and not a proper case of constraint that should be expressed for the whole component. For instance, a statement establishing that all endpoints of a component must be used only with a particular protocol would be a valid case of constraint.
While those concerns are properly architectural, only μTOSCA, thought inheritance of the TOSCA specification, allows to express them for instance by setting ranges for values of numerical attributes, or imposing elements of certain types in groups thus transmitting the service constraints to architectural gatherings. We see that as a deficiency as constraints give a proper architectural information on components. Some simple constraints could be easily expressed such as a rules on the number of connections, on usable technologies, on communication protocols, on position in larger configurations or on the execution environment. Executional constraints such as response time or connection limit are also possible to express and couple with monitoring tools.

The "Implementation" column refers to the ability to give indications about how the described component should translate into an program. For the languages that provide such a possibility, we must distinguish tree categories.
The first category consists of the executable languages. Jolie, Compose (through Docker Compose) and Kubernetes have an execution tool that allows to immediately execute the system. μTOSCA and Kompose have tools to compile their application descriptions files into Docker Compose files and Kubernetes manifest and provide means to directly set properties in resulting files. Obviously the implementation category becomes either irrelevant or validated by default for them. We can note, that even if executables, the expressive power of the language sometimes excess the executional power of the related execution tool. For instance, it is possible to express a number of replica in Compose while the Docker Compose platform does not have a mean to use that information [23].
The second category are the languages that allow to associate technologies with components. In LEMMA, it is possible to define technologies in a particular viewpoint that can the be placed as an additional layer upon described microservices. This applied technology then implies some precisions to be added to the microservice description for the description to be correct and the code generation to be performed. Other technologies, such as infrastructural microservices or databases can also be defined in the technology viewpoint files and then only referred to in the deployment section of the application. JHipster does not let the user define custom technologies but includes them in the description of microservices. Some arguably infrastructural responsibilities (e.g. data

storage, authentication...) are then taken care of by resulting microservices invoked at execution time.
The last category includes the languages that allows to directly write target code which is obviously some sort of implementation indication. MicroBuilder demands the user to precise the Java code of non implicitly generated methods and LEMMA in its deployment section allow the user to introduce some Docker code. The aim of an architectural language is to provide an higher level view of the system and since procedural code often needs deep analysis to understand its ramifications its introduction in architectural description is not to be generally considered desirable. However in the case of those languages that are directed toward code generation and use the introduction of executable code in a limited and well-defined manner it limits the discrepancy between the architectural description and the target executable code that is generated. Moreover those precisions limit the ulterior refinement work.
Silvera does not includes concepts dedicated to implementation and MicroArt being used as a language to only describe existing architectures has no need for it.
In summary, we have seen that most languages are either directly executable or provide some way of giving implementation directives. The fact that this area is well covered is to be put in touch with the fact that most of these languages aim to be used for code generation.

3.4.2 Connectors and Configurations

Connectors and configurations are considered essential parts of an ADL (see Section 3.1) however they are absent from most ADLs and when not, their presence is limited. Table 3.2 shows their presence in the presented languages.

Language	Connectors	Configurations
LEMMA	N	N
Silvera	N	N
MicroArt	Y	N
Jolie	N	Y
MBuilder	N	N
JHipster	N	N
μTOSCA	$\sim$	$\sim$
Komponents	Y	N
Compose	N	Y
K8s	N	Y

Table 3.2: Presence of connectors and architectural configurations in in microservice ADLs.

The "Connector" column indicates whether the language has a notion of connector or not. As discussed before, interfaces are to be distinguished from connectors as they define the points on which the connectors are meant to connect to. Through that specification they can give limitations about which kind of connectors can be used but connectors must be independent entities representing a communicational link between two components or more. Because of that, explicit dependencies written in components descriptions are also not connectors. In a fully and well described system they would imply the existence of a connector that could render their specification unnecessary.
That being clarified we now consider the presence of the connector concept in existing ADLs and remark that it is almost non-existent. MicroArt provides a rudimentary concept of connector under the name of "Link" that is created by offline and online analysis of the system and consisting only in the source and destination of the connection. As MicroArt does not have a concrete syntax it serves only as an intermediary to produce the dependency graph of the application. The only example of a connector independent from service definition in a concrete syntax is found in Komponents description with a separate section of the description allowing for connectors of different cardinality for source and target.
μTOSCA comes close to having a satisfying implementation of connectors through inheritance of the `relationship` type from TOSCA and its sub-typing capabilities but does not make them first class citizens of the language. Jolie has a concept named Aggregator, discussed in the next paragraph, that comprises some parts of what a connector is such as having input and output ports corresponding to other microservices as well as protocol specification. However a Jolie aggregator does much more than what a connector is expected to do such as protocol transformation, and unified response construction from different microservices. It also manifests itself as an independent process at runtime and so we fell that it did not fit into the connector category that is meant to represent immaterial relations between components.

Connectors not being addressed independently can be considered an important deficiency in existing microservice ADLs as without them the description of the application tends to be a collection of unorganized elements and thus miss the exhibition of the architecture. On the other hand, explicit sections dedicated to them tend to be difficult to make sense of, and considerably weight on application description length.

The "Configuration" column refers to the possibility to express architectural configurations in the language. As it is the case with connectors configurations are an uncommon feature of microservices ADLs.
LEMMA, Silvera, MicroBuilder and Jhipster have no concepts that can be considered as configurations. MicroArt has a notion of "cluster" in its metamodel but makes no use of it as no information extracted from the offline or online analysis of the application is mapped to the concept. We then considered that it did not provide architectural configurations.

μTOSCA allows the gathering of services in `groups`, a concept inherited from TOSCA. It is a reasonable way to highlight policies and even patterns through the use of type constraints. However it lacks a proper set of attributes, does not add logic to the services it contains and does little more than applying the constraints defined for services. Because of that we do not consider them a fully satisfying implementation of configurations.
Komposes comes close to having a real concept of configuration through the `composite` concept that allows to make reusable composition of microservices. However parametrization of `composite` objects is difficult, as it is done indirectly through importation in a newly defined object. The `composite` concept has much of what a configuration concept needs but we consider that it cannot considered as such because each service can only be part of one `composite` making impossible to express different patterns affecting a single service. Jolie has two concepts that can be considered architectural configurations: aggregation and embedding.
Aggregation is used to orchestrate groups of services that produce different responses and produce a single composed response to the caller. It describes a composition of microservices but has an existence of its own, both logically and at runtime as an independent process, as it performs actions that are not limited to the piping of microservices communications. However its primary function is to organise the workflow of existing microservices describing, the logic tying them together and can thus provides a higher level of abstraction in which the aggregator is seen as a component to its immediate environment. Because of that we consider aggregators to be architectural configurations.
Embedding is a concept in Jolie that allows to combine, from an execution point of view, existing services into a new, bigger one. Embedded services are then parts of a larger service and not existing anymore as independent processes but as part of a new service that usually add unifying logic on top of them. That part can make it arguable that embedding is not an architectural configuration since it does more than describing the system. We consider that if we dissociate the language from its implementation (i.e. current interpreter) the added logic can be seen as a description of what's happening in its parts as the embedding concept can be used as a wrapper around the contained services. Because of that the embedding concept is not only fitted to be considered a configuration but also the more advanced example of configuration we have encountered as it allows to define a custom logic for services interactions. Compose proposes a series of keywords that affect the entirety of the system. The "networks" keyword creates a communicational space shared by default by all the services of the system or by a selected number of them that register to it in their description. Multiple networks can be defined creating communicational cluster of services. In the same manner, the "volumes" keyword defines a cluster of services sharing a data storage location. That ability of clustering given by Compose makes it so that it can be considered as featuring builtin specific architectural configurations.
Kubernetes offers resources such as Ingress, Services, Deployments and Pods can

be seen as architectural configurations as they allow to cluster services. Ingress allows to route incoming communications toward different kinds of pods depending on the path, it thus defines a cluster of pods accessible from the outside of the application. Deployments are usually used to manage one kind of pod but can refer to multiple pods. Services create a communicational cluster for different pods. Pods can be constituted of multiple pods, in practice usually a main service and its dedicated services such as a database or an initializer. Doing so it produces a composite.

Surprisingly enough, the executable languages, whose main purpose is not the description of the architecture, are those that contain architectural configurations that are arguably the highest degree of architecture description. That is partly explainable by the fact that architectural configurations are essential for those languages to perform while it appears as superfluous to other ADLs. We consider that as a deficiency, even more than we did with connectors, since the architectural configurations are a condition to make sense of the system in a global manner regardless of the viewpoint (e.g. communicational, logic, resources...).

3.4.3 Other Aspects

Table 3.3 gathers other aspects of microservices systems that can be found in multiple ADLs in different forms.

Language	**Design** Patterns	**Data** Structures	**Deployment**
LEMMA	N	Y	Y
Silvera	Y	N	Y
MicroArt	N	N	N
Jolie	N	N	∼
MBuilder	Y	Y	N
JHipster	Y	Y	Y
μTOSCA	N	N	Y
Komponents	Y	N	Y
Compose	N	N	Y
K8s	N	N	Y

Table 3.3: Presence of other microservices systems aspects in microservice ADLs

The column "Design Patterns" indicates whether or not design patterns appear in one form or another in the language. An extensive list of design patterns and their descriptions can be found in Section A.

Compose and Kubernetes do not have a notion of design patterns although some of their structures suppose implicitly the presence of tools whose functioning implements some of the more usual patterns. The "network" concept in Compose and the "service" concept in Kubernetes for example imply the presence of the "registry pattern" and of the presence of service discovery through DNS. However these languages do not refer explicitly to those patterns which is what is expected of an architecture language representing them. The Ingress component of Kubernetes can be seen as acting like a gateway but describes rather a microservice than the architectural impact of a gateway pattern as it does not encompass microservices indirectly accessible. Lastly if the Pod structure allows to create some topologies which correspond to design pattern (e.g. database per service), it does so implicitly, by having a description that happens to correspond to the pattern but is not identifiable as such in itself and is no reusable.

Jolie happens to fall in the same category as its topological structures are not parametrizable enough to express a design pattern in different applications. It is possible to express a particular topology in an application that would fit a given design pattern but it is necessary to include all the possible services that can be used and apply conditionals to determine which one of them is used in this particular instance of the composite structure.
Kompose, LEMMA and MicroArt have no structures or keywords to indicate the usage of a design pattern. However Kompose presents multiple example of pattern implementations. The `composite` type is used to create reusable structures composed of multiple elements but the limitation of having services defined exclusively in one `composite` make it impossible to represent their interconnections.
Silvera, MicroBuilder, JHipster and μTOSCA all provide references to well known design patterns in their components' descriptions. In Silvera, when describing a microservice you have to explicitly reference its registry, implementing de facto the "registry pattern". MicroBuilder and JHipster both couple databases with functional microservices implementing de facto the "database per service" pattern but do not give the option to opt for a "shared database" pattern. μTOSCA has tree keywords (`dynamic_discovery`, `circuit_breaker` and `timeout`) used in the `relationship` section of service definition signifying the use relational patterns between two services. The `Edge` group gathering the services with ports open outside of the application can also be considered a pattern. Through `groups` new patterns and policies can also be added.
In summary design patterns are not widely used in existing ADLs for description of microservice applications. Most of the languages use design patterns implicitly in the sense that their toolboxes either generate applications with fixed patterns or that the tools used to execute the code, in the case of executable languages, rely on them. That situation can be seen as pragmatic but is also a loss in generality as it limits the range of architectures those languages can describe. The best situation would be to

dispose of structures allowing for the creation of reusable, parametrizable architectural configurations whose particular forms would match design patterns. Unfortunately no existing microservice ADL provide such a concept in its concrete syntax.

The column "Data Structures" indicates if a language allows to describe the data structures existing in the application.
As Compose and Kubernetes are not involved with the functional code, they have no mean of describing data structures linked with microservices. μTOSCA and Komponents being also deployment oriented they do not provide ways to represent data structure although μTOSCA has a `DataStore` microservice type. Jolie possesses a series of inbuilt methods that allow to communicate with a number of database technologies in a standardized way but do not allow to define persistent data structures owned by microservices. Being a programming language it allows for "runtime" data structures to be defined in the microservices code.
LEMMA, MicroBuilder and JHipster have a strong focus on data structures used in the system and are the only languages we studied that allow to explicitly define them. All three languages allow to define databases owned by microservices de facto implementing the "database per service pattern" and not giving the possibility to describe architectures with shared databases.
Microbuilder does it in the most straightforward manner by simply having the data structures defined in microservices' descriptions. It provides a distinction between data structures stored in an independent database whose persistence does not depend on the microservices life as a program and data structures that exist for the lifetime of the microservice and that are lost when it stops its execution.
JHipster defines data structures separately from microservices. The microservices can refer to a number of data structures. Each microservice referring to a data structure has to be considered as having its own instance of the data structure, effectively existing as a dedicated database whose particular technology implementation is determined in the description of the microservice. A specificity of JHipster regarding databases is that it allows to describe relationships between databases.
LEMMA is the language where the data concern is the most developed. Data structures are described in their own section section of the language named "data viewpoint". The main distinction is made using the domain-driven design of "entities" and "value object" who represent respectively persistent data structures and transient data structures. The data structures' definitions can then be used in the microservice viewpoint. Entities are typically objects in databases dedicated to a microservice which implements the "database per service" pattern. Where LEMMA stands out is in allowing do define precisely through value objects the volatile data structures used for requesting and answering requests. Through its data viewpoint, LEMMA addresses the data communication concern in the most precise way we have encountered.

In summary, there are different examples of data structure representation in microservices' ADLs focussed on defining the databases dedicated to specific microservices. While it is not as specific to microservices as the "database per service" pattern is, "shared database" pattern is not specifically describable in any of the languages which is a limitation in terms of architectural description.

The column "Deployment" indicates whether or not the language allows to insert descriptions relative to the deployment of the system.
Compose and Kubernetes are runtime environment configuration oriented so they do this by design. Although μTOSCA and Komponents are technology agnostic, they compile toward Compose and Kuberetes and some of their properties are directly mapped on those target languages properties. They also offer possibilities to directly set properties of any deployment language they could be compile to.
Jolie and Microbuilder have not a deployment section anywhere in the architecture description. The closest indication relative to deployment is that in both of those language the exposed port is specified in the microservice description. Jolie adds the location (url or ip address) and couples deployment configuration files to the main system description.
JHipster and Silvera both have a "deployment" keyword used in the scope of the architecture description to parametrize the deployment. JHipster uses either Docker Compose, Kubernetes or Open Shift for deployment and it uses its "deployment" description to choose between those three technologies and configure them using specific fields for each of them. Some technology agnostic keywords such as "number of replicas", deciding between "ephemeral" or "persistent" storage allow to indicate some general notions about how the system should be deployed. In JHipster, multiple deployment blocks, for different technologies, can be defined at the architecture description root, with each of them ultimately producing deployment for the whole system. Silvera can only produce a Docker Compose files at generation time but defines the deployment block inside microservices description, allowing to have specific instructions for each of them. It expresses the deployment in a more technology agnostic manner making no references to the particular technology it uses in the deployment block. The most determining fields that can be defined are "port", "replicas" and "restart policy". It is to be noted that it is possible to specify in the "host" field if the application is intended to run in a "physical machine, virtual machine or container".
LEMMA as for the data concern, dedicates a whole viewpoint to the deployment concern named "operation viewpoint" that is a part of the architecture description expressed in dedicated files. Services are imported either from the "service viewpoint" for functional microservices, or defined directly as infrastructural services using technologies defined in the "technology viewpoint" files. Each of these kinds of deployment units specify exposed ports and use a deployment technology defined in the "technology viewpoint" files that allows to specify specific fields to fill (e.g. image, dockerfile). The infrastructural

deployment units have specific fields allowing to express on which infrastructural nodes they are depending and which functional services depend on them. They also permit to specify some technology specific fields (e.g. database password).
In summary, set aside deployment configuration languages, deployment is a common feature in microservices ADLs but stays on the surface leaving the specificities to be modified in the necessarily existing or generated deployment files.

3.4.4 Decomposition

Languages have fundamental elements of decomposition as defined in Section 4.2.2. Table 3.4 compare identified microservice ADLs in that regard.

Language	Decomposition	Focus
LEMMA	microservice	interface, data
Silvera	microservice	interface
MicroArt	microservice	interface, dependencies
Jolie	microservice	interface, logic
MBuilder	microservice	interface, data
JHipster	microservice	data, technology
μTOSCA	microservice	configuration
Komponents	microservice/composite	configuration
Compose	microservice/container	configuration
K8s	workload	configuration

Table 3.4: Main decomposition and focus of microservice ADLs

The "Decomposition" column represents the kind of components the different ADLs use to decompose the application and the "Focus" column indicates the main axis of description of that component. Unsurprisingly in most of the cases the base system brick is the microservice. Here we distinguish microservice from container or pod, respectively the unit of decomposition for Compose and Kubernetes. "Microservice" designates the program and in all cases but JHipster, the microservice is first described through its interface and without any technology specification. Under that aspect of component description which is necessary to any architectural description, different focuses are brought by the different languages. Data description provided by LEMMA, MicroBuilder and JHipster is a key aspect of microservices as it poses specific challenges to preserve their independence. Jolie being a programming language has as a main focus to describe the microservice logic which is entirely contained in its description.
JHipster is a particular case as it does not explicitly provide endpoints. The "application" structure, that in a microservice context refers to a microservice, acts as a blueprint,

gathering and configuring different technologies taking part in the final program. We considered that it is core to JHipster as it constitutes the microservice description. On the other hand in LEMMA the technology model applies as an overlay to the technology agnostic service description.
While to some extent the container and the program executing can both coincide as being the "microservice" the distinction is to be made when considering the focus of a language. Indeed μTOSCA, Komponents, Compose and Kubernetes use as base element not the program but the environment in which it is executed. In accordance with that decomposition the description sets the properties of the inside of the components, where the program is just a particular attribute among other such a environment variables, and the edges of the components with properties such as ports or allocated resources.

3.4.5 Aims and Targets

Listing 3.5 shows that the presented languages have different aims and target different stakeholders. Most of the languages have for ultimate aim to produce the application executable code or at least a code base to expend in order to make the application executable. They offer a global view which is of interest for architect and produce code that is of interest for developers. However, as we've seen before, because they do not offer abstraction to express and abstract the architecture,they are more suited for small applications where those two roles have a good chance of being assumed by the same persons. MicroArt can be put aside as an original proposition targeted purely toward architects aiming at understanding an application architecture. μTOSCA, Komponents, Compose and Kubernetes are not following the same aim as they are closely linked to deployment and operation technologies thus having the advantage of being immediately useful the last two being immediately interpreted by their related execution software and the first two being mapped to them through their toolbox in a direct manner needing no further modification.

Language	Aim	Target
LEMMA	code generation	architects/developers
Silvera	code generation	architects/developers
MicroBuilder	code generation	architects/developers
MicroArt	dependencies reconstruction	architects/developers
Jolie	development	architects/developers
JHipster	technology blueprint	architects/developers
μTOSCA	operation configuration	architects/operators
Komponents	operation configuration	operators
Compose	operation configuration	operators
K8s	operation configuration	operators

Table 3.5: Principal aims and stakeholders targets of the identified solutions

3.5 Studied Microservice ADLs : Toolboxes

Toolboxes are fundamentally linked to the languages that describe an architecture, often they are the reason the language was created in the first place. It is particularly obvious for Compose (through Docker Compose, its main implementation) and Kubernetes for which the language and supporting tool are so perfectly aligned that in practice users make little difference between the two.
The tools provided can be of very different natures. We grouped tools that are relative to implementation, analysis and visualization as these categories appeared as the most important.

3.5.1 Implementation Tools

Table 3.6 presents the tools that are relative to execution in the studied ADLs.

Language	Executable	Generation
LEMMA	N	Y
Silvera	N	Y
MicroArt	N	N
Jolie	Y	Y
MBuilder	N	Y
JHipster	N	Y
μTOSCA	N	Y
Komponents	N	Y
Compose	Y	Y
K8s	Y	N

Table 3.6: Presence of implementation tools for identified solutions

The column "Executable" refers to the existence of a tool to directly run described architecture without compiling it to an existing, executable, target language. Three languages fit that categorization: Jolie, Compose and Kubernetes.
Jolie is a procedural programming language dedicated to create microservices systems and so it comes with a specific interpreter that has been realized to run instructions (e.g. parallelism) or producing behavior (e.g. message transmission error) designed to be particularly relevant for microservices architectures. Whether the system is run using a container technology or the jolie command, each service is a different process running a different instances of the Jolie interpreter. The Jolie interpreter is written in Java so it relies on the Java Virtual Machine to run.
μTOSCA and Komponents are not directly executable but have compiler to both Compose and Kubernetes allowing exact execution of their description files without modification.
Compose files are executable with the docker-compose tool that translate those declarative files in a docker application that meets the described properties. In the end, the tool associated with the compose description is then docker and from a point of view compose acts as a wrapper around docker, avoiding the pains of scripting or command typing to obtain the desired system. As Compose relies ultimately on Docker, it benefits from all its features such as service discovery, network communication and isolation or security features. It also adds functionalities on the system such as scaling.
Kubernetes' files, known as manifests, are intended to be used to interact with a Kubernetes running system that manages resources, taking away the infrastructural concern of deciding where and how to set up the modifications, from the user. Setting up the Kubernetes system has a relatively high initial cost but once its done the infrastructure is made completely abstract. What is left for the user to do is simply to describe what it desires and transmit it to the Kubernetes interface that will take the appropriate measures to make so that the real state of the running application match

the description. Kubernetes is often refered as an orchestrator, which is appropriate if we think of orchestration as dealing "with how different services are composed into a coherent whole"[3]. Kubernetes fits that broad and blurry definition by taking charge of a multiplicity of strong requirements for a microservice application such as service discovery, scaling (both vertical and horizontal), message routing and many other aspects. It also fits the more precise definition of an orchestrator specifying that "in particular, it specifies the order in which the services are invoked, and the conditions under which a certain service may or may not be invoked¨[3]. It is to be precised though that Kubernetes does take charge of that aspect from the infrastructural point of view, for instance through load-balancing and routing, but does not get into the control flow of application logic which was historically the initial subject of orchestration. However, to be effective the described Kubernetes system description must match the application logic.

The column "Generation" refers to whether or not the corresponding microservice ADL is coupled with a tool that can generate code in a language that already disposes of the tools needed for it to be executable (e.g. an interpreter, a compiler to machine code). MicroArt and Kubernetes have no tool for generating code in an executable target language. MicroArt is intended to be used in architecture reconstruction and provide information about microservices dependencies in the system; converting the extracted information in an existing executable language would have been a totally different purpose and challenge. Kubernetes relies heavily on its own specific structures and is designed to be coupled to its particular platform which makes it difficult to translate. Also it is fairly recent and arguably still the last major deployment and operation technology that has arisen so the need for translation might not exist yet.
Jolie and Compose are executable languages (meaning that they dispose of a tool to directly run the code) but can also be translated respectively to Java and Kubernetes. Jolie furnishes a tool named "jolie2java" that permits to convert a Jolie service to Java. Jolie furnishes Java libraries to allow easy compatibility between Jolie and Java services, when using jolie2java, the Java services produced are then able to directly invoke the Jolie services' operations. Compose has a tool named "Kompose"[58] used to convert Docker Compose files to container orchestrators, and in particular to Kubernetes. It is an official Kubernetes project. It generally requires little modification to the produced files to have a working application and allows to complete the Docker Compose file with custom labels to obtain the desired output and limit manual refinement. It is to be note that as Kubernetes implements more functionalities than Docker Compose, some microservices might have been initially designed to take care of task that are native to Kubernetes and thus resulting in a non optimal architecture where specific Kubernetes resources are not taken advantage of and even acting as redundant parts of the system.
LEMMA, Silvera, MicroBuilder and JHipster are mainly oriented toward code generation. They all provide means to generate Java code either through a command line compiler

tool (Silvera, MicroBuilder, JHipster) or as an Eclipse plugin (LEMMA). Silvera, JHipster generate code using templates in a model-to-text transformation, while MicroBuilder compiler named "MicroGenerator" transforms the system description in a direct model-to-text transformation. To clarify, the description of the system is the model that is transformed to text, that transformation is done either directly or disposes of a modular component, a template, that will determine through place holders replaced at compile time what the final code will be. LEMMA has an additional initial step to this process, converting the system description in instances of builtin metamodels (mapped on the different viewpoints in the language) that can then be used in a model to model transformation (effectively toward a general object oriented language metamodel) before being transformed to code. That supplementary step allows for a lot more flexibility and possibilities such as model analysis and permitting the creation of transformations tools to other programming paradigms metamodels. Ultimately in each of these solutions, a refinement phase is needed after code generation.
μTOSCA and Komponents provide compilers, respectively μFreshener and Deployer, producing exact mappings of their description in Compose and Kubernetes, allowing deployment without modifications.

3.5.2 Analysis

Table 3.7 represents the tools available for the different languages we studied. It is largely an unexplored territory as only Silvera and MicroArt have some analysis capabilities.

Language	**Offline**		**Online**	
	Constraints	**Metrics**	**Constraints**	**Metrics**
LEMMA	N	N	N	N
Silvera	N	Y	N	N
MicroArt	N	N	N	∼
Jolie	N	N	N	N
MBuilder	N	N	N	N
JHipster	N	N	N	∼
μTOSCA	Y	N	N	∼
Komponents	N	N	N	N
Compose	N	N	N	∼
K8s	N	N	N	∼

Table 3.7: Presence of architectural analysis tools for identified solutions

Silvera is the only language having a tool to establish an offline evaluation of the system; it does so by calculating metrics at compile time, in a phase preceding

code generation. The six metrics evaluate mainly the independence of microservices in order to draw attention on potentially critical or problematic parts of the system. Each microservice is subject to evaluation for each metric, this process resulting in an application score (i.e. average of all microservices scores) for a particular metric.
μTOSCA benefits from the existing TOSCA engines, e.g., Cloudify [10], xOpera TOSCA orchestrator [108], to enforce the constraints expressed in the application descriptions.
MicroArt through online analysis is able to extract microservices dependencies. However it does not provide a comprehensive way of presenting the gathered info, leaving the real analysis part and deduction making to the user.
Online analysis is relatively unexploited. Kubernetes natively produces metrics about resources used by pods on the underlying infrastructure hosting them, resources used by the kubernetes system itself and custom metrics exported by applications. In the same manner, Docker produces resources metrics accessible through the API. Since those platforms control the execution of the application it is not surprising that they produce metrics about resource usage. However these raw metrics are not correlated to the architecture of the microservice application and thus do not produce an evaluation of the system.
A surprising observation is that, apart from JHipster, generation oriented ADL do not automatically integrate technologies to produce metrics. JHipster at compile time integrates components to produce resources usage metrics (Prometheus), ways to exports application specific metrics (Logstash) and to produce a graphical representation (Grafana).
Analysis is a particularly untouched territory which is explainable by different factors. Before thinking about having a tool to verify constraints it would be desirable to dispose of ways to express constraints about a microservice system. If such an expression capacity was existing in a language, matching this syntax with an existing, general, model analysing tool (e.g. Alloy [2]) could prove less tedious than one could anticipate.

3.5.3 Visualization

Visualization is the most effective tool to represent high level informations. It is useful to share informations between different domains, but also for domains specialists to have an easy reminder of the system and its environment. Table 3.8 presents the visualization tools present in the identified ADLs toolboxes.

Language	Representation	Abstraction	Views
LEMMA	N	N	N
Silvera	N	N	N
MicroArt	Y	N	N
Jolie	N	N	N
MBuilder	N	N	N
JHipster	Y	N	N
μTOSCA	Y	Y	Y
Komponents	N	N	N
Compose	N	N	N
K8s	N	N	N

Table 3.8: Presence of visualization tools for identified solutions

MicroArt is oriented toward architecture reconstruction and uses a graphical representation to represent the system it analysed. In the representation, microservices are represented as a collection of boxes, one for each identified endpoint with a section for exposed endpoints and another for required microservices. Those boxes are connected with lines indicating a dependency. As MicroArt does not provide a concrete syntax, it is fair to say that the graphical representation is the actual language and not a graphical representation of it. MicroArt does not provide different levels of abstractions or views of different aspects of the system.

JHipster provides a very simple representation of the different microservices as boxes containing the different "entities" and their relations. The visualization is generated directly as the description is being written and even though it is rudimentary is shows the power of a graphical representation as local modifications appear immediately and clearly in their relation with other preexisting elements. The visualization provided by JHipster could be considered a view as it describes only the data structures. Through its IntelliJ plugin, JHipster also generates UML representation.

Komponents has a modeler tool to represent descriptions with a graphical syntax mapped to the language. It allows to import Komponents files to view their representation and then modify the application before producing the corresponding text description.

μTOSCA, through μFreshener, provides the most complete graphical tool of the toolboxes associated with the languages studied in this thesis. It is able to show different levels of abstractions, from detected patterns to components descriptions, and view such as communication view and deployment view. Its main goal being the identification of anti-patterns, if also highlights, proposes modifications presented in a graphical way that can be interactively compared to the original architecture.

The usefulness of those examples shows that visual representation is a missing feature in many ADLs that could be an important and relatively easy tool to integrate. Producing a mapping from the ADLs to existing technologies, such as the C4 model[98] that already

provides a way to represent different abstractions level, could be an effective way to do so.

3.5.4 Other Tools

Table 3.9 reassembles different tools used to help developers and architects understand and improve the system.

Language	IDE	Active Specification	Architecture Reconstruction
LEMMA	Y	N	N
Silvera	N	N	N
MicroArt	N	N	Y
Jolie	Y	N	N
MBuilder	N	N	N
JHipster	Y	N	N
μTOSCA	N	Y	Y
Komponents	N	N	N
Compose	N	N	N
K8s	N	N	N

Table 3.9: Presence of other tools for identified solutions

The column IDE indicates if a microservice language either provides a dedicated IDE or plugins for existing IDEs.
The most usual way to provide IDE support for ADLs is to furnish a plugin for an existing IDE. The languages that do so are JHipster (IntelliJ), LEMMA (Eclipse) and Jolie (Visual Studio Code) with all of them providing syntax highlighting and text completion. JHipster is the only language to propose a dedicated IDE accessible through its website. Apart from μTOSCA, active specification is not part of any toolbox which is to be expected since static analysis is almost non existent. Most of the systems we have identified make strong decisions that can be considered as good practice for the type of application they aim but it cannot be considered active specification as it does not adapt to the particular architecture being described. Positive active specification is particularly challenging as it needs not only to understand the described architecture but also know of the relevant architectural compositions through either an exhaustive list or general rules. The tool must then be able to correctly identify an unsatisfying architecture and a better proposition to present the user with.
Active specification can be seen as the pinnacle of user support, but its implementation as a positive proposition is difficult to achieve. An easier implementation is the detection of anti-patterns and the proposition of a solution. We note an article presenting a proof of concept using LEMMA to identify and correct architectural anti-patterns has been

published[105] that could lead to a future implementation.
Architecture reconstruction is the main focus of MicroArt which focuses particularly on endpoints dependencies through static and online analysis. MicroArt architecture reconstruction tool first analyses the source code then performs an online analysis to ultimately ask the user for a refinement phase where the service discovery microservice is identified manually and removed. As it acts as an intermediary the tool links the identified calls to the service discovery microservice to the corresponding calls from the service discovery microservice and reveal the logical dependencies in the system.
Both of those tools are the main focus of μTOSCA. Its toolchain is constituted of two mains tools. μMiner automatically reconstructs the application architecture from Kubernetes manifests by static analysis and optionally online monitoring. From there it produces μTOSCA descriptionof the application instantiating the patterns implemented in the language. Other information such as non implemented patterns or anti-pattern are stored in side files. Those files are transmitted to the second tool μFreshener, the active specification tool, that provides a graphical representation of the application and suggests modifications to improve the application architecture.
Different techniques have been explored in academic papers to reconstruct the architecture such as log collection, packet interception and code analysis. It is also a large area of interest in the industry due to the ever growing complexity of Microservice applications[8]. However they focus on the particular concern of microservice dependencies representation and are not coupled with a language allowing to describe the system in its different aspects.

3.6 Synthesis and Research Challenges

Through the analysis of existing microservices ADLs, we have identified several limitations that are not covered and that motivate the research questions introduced in Section 1.2.

One of the main conclusions is that existing solutions, having as first objective to produce or be runnable code, are not oriented toward describing the underlying structures and patterns that are properly architectural. This limitation is particularly perceptible in the lack of structures to describe architectural configurations. When such structures exist, they are either tied to fundamental necessities of an application execution environment, as in Compose, or specific to the tools to which the language is destined, as in Kubernetes. Jolie constitutes a partial exception, but the architectural configurations it provides are thought as executional units, adding logic instead of describing the one that is already present, and they lack flexibility in terms of reuse.

As a consequence, microservice applications are still described primarily as lists of microservices, with architectural logic remaining implicit. This situation limits understandability and hinders the expression of global concerns, as concerns are scattered

among multiple elements of the system description, making modification and evolution error prone.

Another major challenge concerns the alignment between architectural descriptions and existing systems. Maintaining a distinct architectural language introduces a constant workload and increases the risk of discrepancy between the description and the actual state of the application. While existing ADLs address this issue through executability or code generation, these approaches generally imply manual refinement phases that rapidly render the initial description outdated. Executable languages further face adoption issues due to the maturity of existing microservice ecosystems, which limits their architectural reach in practice.

These observations highlight the need for architectural description approaches capable of expressing meaningful architectural structures while remaining applicable to existing systems, ideally through the extraction of architectural information using static and dynamic analysis or at least through simple but powerful concepts and syntax reaching the fundamental architecture structure of applications with minimal verboseness.

3.7 Conclusion

Microservice architecture brings solutions to limitations of earlier software architecture styles by promoting autonomy, technological diversity, and fine-grained scaling. These benefits stem from the independence of microservices, but this same independence also increases application complexity, leading to difficulties in understanding the system and managing its evolution.

Architectural decisions therefore play a determining role throughout the application lifecycle. However, the distributed and heterogeneous nature of microservice-based development increases concern scattering and makes it difficult to obtain a unified view of the system.

Architecture Description Languages aim to address these difficulties by providing a unified vision of a microservice architecture through the description of components, their relations, and architectural configurations, supported by analysis and visualization tools. Nevertheless, the state of the art reviewed in this chapter shows that most existing microservice ADLs remain oriented toward component description rather than architectural structure. This approach constitutes the sharing of a common underlying model limiting architecture description that is described in the next section.

These limitation show the unsatisfying state of Microservices ADL, answering the first research question described in Section 1.2 and motivating the proposal for an alternative model.

Chapter 4

A concern oriented architecture description language for microservice applications

Having established the shortcommings of existing ADLs in Chapter 3 we dedicate this Chapter to first examplify those findings with well known use case. We then propose our solution consisting in the extension of the microservice architecture description paradigm with the Concern concept resulting in the COMSA language. We use literate programming [57] to present the Concern concept as well as the constructions of the COMSA language.

4.1 MSA common metamodel shortcomings

4.1.1 Compose: a relevant example of the MSA common metamodel

Figure 4.1 represents the identified tacitely accepted common model for existing microservice ADLs and is a limiting factor in the gains induced by the adoption of MSA. Those shortcomings are rooted in the base component for a microservice application description that is the microservice itself.
The model represents a Microservice application that is composed of microservices. The microservices are the base components of the system and the main first class citizen of the language. Those microservices are composed of properties that can be of different natures depending on the overarching aspects of software architecture the language focuses on.

The Compose specification[22] is a good example of the common MSA model. It falls in the category of configuration languages for microservices and in particular in the deployment configuration category. Although other entities than microservices can be defined such as networks, or volumes, those are not necessary as they can be defined directly in the microservices descriptions to obtain an equivalent

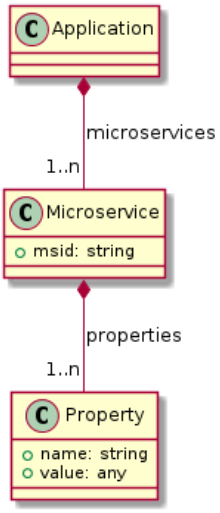


Figure 4.1: MSA classical metamodel [70]

description. The only required section is the "services" section that covers all the configuration capabilities of the language and thus we are justified in considering that the metamodel that underlies the Compose language strictly fits the one pictured in Figure 4.1.

It is to be noted that the properties that constitute a service description in Compose are relative to deployment and resource allocation for a particular execution technology that is the container technology. However the Compose properties are largely the same or equivalent of those of other execution technologies such as virtual machines or simple executables on a host.
In the same way, while some Compose concepts are relative to the natural execution software target (i.e. Docker) they are either not central or particular instances of a more general concept.
The most obvious contrary example would be the Kubernetes manifest language, who's structured around technology specific concepts that while translatable have not equivalent in other languages.

The high generality of the Compose language, its weak ties to a target execution software and its established popularity through its yaml implementation make it a prime candidate for investigating the common MSA model.

4.1.2 MSA common metamodel shortcomings through an example: the Teastore application

The Teastore application [56] has been developed for benchmarking, testing and more generally to demonstrate microservice architecture. It follows best practices on a minimal example, has been referenced over than 100 times in scientific papers [100] and therefore is an obvious choice to exemplify the consequences of adopting the common MSA

metamodel for a microservice ADL.

Listing 4.1 is the Docker Compose file found in the Teastore repository. Considering the `services` keyword *(Line 2)* as the application root, we can observe the adequacy with the common metamodel schematized in Figure 4.1 having the microservices at the application base level *(Lines 3, 7, 13, 22, 29, 36, and 43)*. Their respective properties can then be found under them.

4.1.3 Understandability

Lack of understandability is the first impression when looking at the file. The listing of microservices does not provide any insight about the architecture of the application. In the same way, the properties are simply listed under them in no particular order.
To make sense of the application expressed in Compose it is needed to rely mainly on the custom names decided by the developers of the application. Most of them are explicit to some degree, for instance the `registry` service can be immediately associated with a well known and widely used technical service of the same name whose core function is to keep track and inform microservices about other microservices addresses. In Docker Compose files, the availability of such basic information relies only on the diligence of the developer.

Impossibility of associating properties listed in microservices related to other microservices is a parallel phenomenon also clouding comprehension.
Among the properties that constitute the description of microservices in a language based on the common metamodel, two groups can be distinguished. First are properties that are architecturally attached to one and only one microservice. A clear example is the `image` property such as `image: descartesresearch/teastore-registry` at *Line 4* of Listing 4.1. This property is distinct from any other microservice. Should it change, for instance because the microservice image was migrated to another repository or because of a version change, no other modification in the file would be needed, assuming that the new target is accessible and equivalent system-wide. It is then a local decision.
The second group however are properties that while located in a given microservice description actually refer to another microservice. An obvious example is the `REGISTRY_HOST: "registry"` at *Lines 19, 28, 35, 42 and 49* of Listing 4.1. The property being located in the `environment` section of the microservices that contain it, understanding of how containers function and of the role of a registry microservice, allow to infer that it refers to the `registry` microservice, evidently being a hardcoding of its designated name for online access to it. Modifying the value is not a local decision as it would necessitate global adjustments, here the modification of the registry identifier, for the application to function.
It is to be noted that this dichotomy is not deductible from the nature of the attributes.

```
version: '3'
services:
  registry:
    image: descartesresearch/teastore-registry
    expose:
      - "8080"
  db:
    image: descartesresearch/teastore-db
    expose:
      - "3306"
    ports:
      - "3306:3306"
  persistence:
    image: descartesresearch/teastore-persistence
    expose:
      - "8080"
    environment:
      HOST_NAME: "persistence"
      REGISTRY_HOST: "registry"
      DB_HOST: "db"
      DB_PORT: "3306"
  auth:
    image: descartesresearch/teastore-auth
    expose:
      - "8080"
    environment:
      HOST_NAME: "auth"
      REGISTRY_HOST: "registry"
  image:
    image: descartesresearch/teastore-image
    expose:
      - "8080"
    environment:
      HOST_NAME: "image"
      REGISTRY_HOST: "registry"
  recommender:
    image: descartesresearch/teastore-recommender
    expose:
      - "8080"
    environment:
      HOST_NAME: "recommender"
      REGISTRY_HOST: "registry"
  webui:
    image: descartesresearch/teastore-webui
    expose:
      - "8080"
    environment:
      HOST_NAME: "webui"
      REGISTRY_HOST: "registry"
    ports:
      - "8080:8080"
```

Listing 4.1: TeaStore's Docker Compose file [19]

For instance an environment variable could also be a local decision. Here the environment variable `HOST_NAME` located at *Lines 19, 28, 34, 41 and 48* in Listing 4.1 do not imply any other modification as a consequence.

Linked properties found in microservice architecture description files are often indicative of communication patterns applications that can be extracted with thorough analysis, coupled with a general understanding of the concepts upon which microservice execution relies on and an extensive knowledge of the specific language used.
That realisation however highlights the fact that while the structure of the application is, at least partly, contained in descriptive files of applications expressed in languages using the common metamodel, **its understanding is reserved to experts who need to allocate considerable amount of time to extract the structure from the text without guaranteeing complete or even correct results.**

Explanatory documentation such as diagrams are then needed to avoid mistakes and to communicate an high-level and simple view to other stakeholders. In [56] the Teastore developers provided Figure 4.2 to give a simple architectural view of the application.

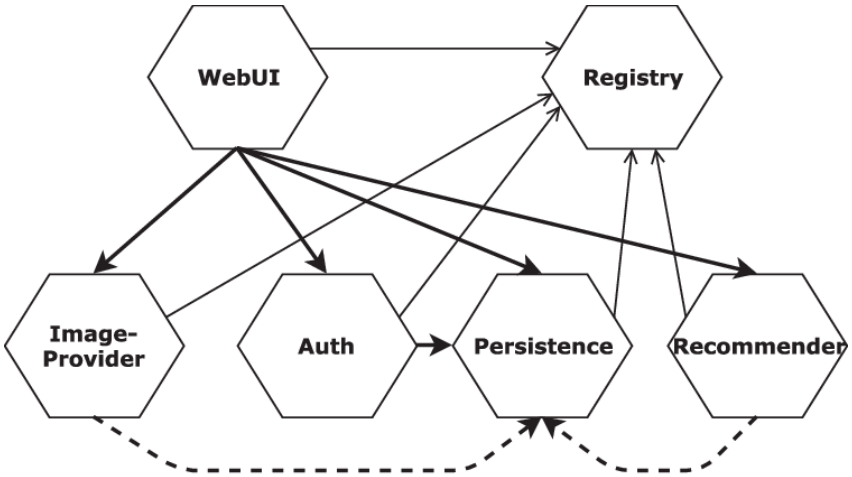


Figure 4.2: Teastore architecture diagram provided in the related article [56].

While it gives a quick and informative view of the application, it brings a number of problems.
Firstly, it is a second document that accompanies the descriptive code. It requires time at creation, but more importantly, it must be updated synchronically to the application evolution. Factually architectural diagrams are not commonly found in application repository, for instance the Teastore diagram is not present in the github repository of the application.
Secondly there is no assurance that the diagram is correct or exhaustive. In Figure 4.2, the `db` microservice that is listed in Listing 4.1 at *Line 7* is not present. Looking at what can be deduced from Listing 4.1, the diagram in Figure 4.3 can be produced. It includes

the link missing in Figure 4.2 but misses other links particularly the link between the frontend and the backend microservices.

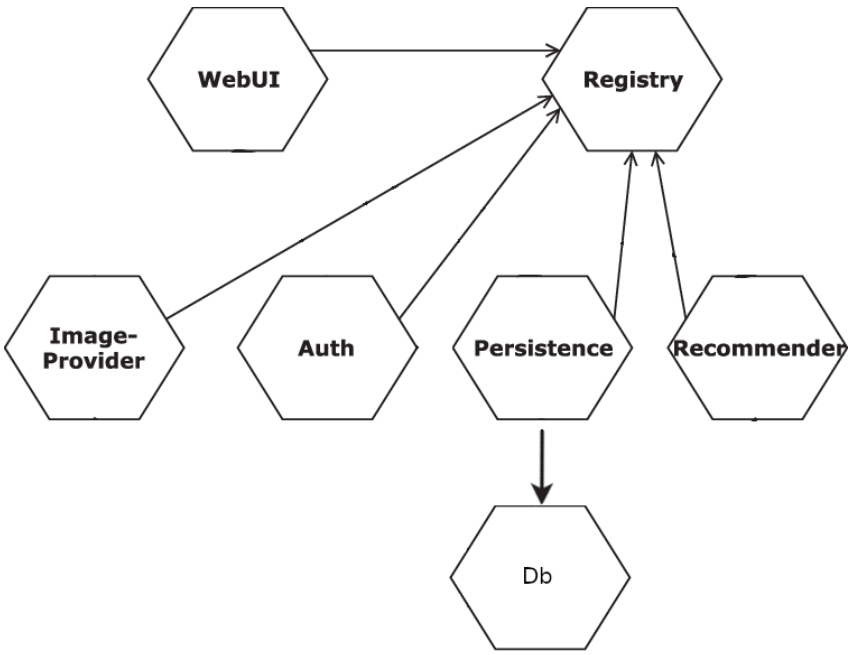


Figure 4.3: Teastore architecture extracted from Compose properties

4.1.4 Coherence and Redundancy

The high degree of redundancy is revealed by merely looking at the Compose file presented in Listing 4.1 and is simply quantifiable.

The `expose` attribute is present and entirely identical in six out of seven microservices, consisting of two lines it represent 23.5 percent of the file. The `REGISTRY_HOST: "registry"` property that is part of the environment attribute of five out of seven microservices is also a strict repetition and represents 9.8 percent of the file bringing the total of strictly redundant properties to a third of the file.

But entirely repeated properties do not cover the total domain of redundancy. In Listing 4.1, `image`, and `HOST_NAME` properties differ only in the microservice identifier part. Those two properties make up for 27.45 percent of the file bringing the redundant properties in the file to more that 60 percent of the file.

Even though we pointed out how it is not the case for the Compose language in Section 4.1.3, the length of the file is not necessarily proportional to how easy it is to understand. Additional writing could as well be obscuring the meaning as it could clarify it, and describing the system in the most frugal way can either be straight to the point of absolutely incomprehensible.

Because of that, from an architectural point of view, it makes sense to consider only the microservice properties and not the specific way they are expressed in the particular language we are using. Using that approach, Listing 4.1 contains 28 microservices

properties. Keeping only one instance of the repeated properties, 19 of them, i.e., a third of properties are redundant and thus possibly unnecessary if the language allowed for factorisation. Listing 4.2 provides a view of Listing 4.1 with the possibly superfluous properties greyed out.
The consequences of such a high degree of redundancy are well known. They include notably error-proneness maintenance difficulties and hindering understandability.

As a symptom, redundancy also indicates a lack of coherence. `REGISTRY_HOST: "registry"` is repeated 5 times *Lines 19, 28, 35, 42 and 49* and as such must be considered as 5 different properties as they configure 5 microservices independently. Indeed modifying the value in one those lines do not affect the others.
For some properties the redundancy can be a deliberate choice as their value is not determined by the architecture. The `expose` attribute for instance is highly redundant in Listing 4.1 but its value could be different for each microservice. We note that its factorisation could also be a sensible architectural choice to indicate a policy of having the same exposed port, materialized as the same property value, for all the microservices. However, this is not the case for the `REGISTRY_HOST: "registry"` property that is indicative of a microservice being part of a service registry pattern and having the identifier of the registry service fixed to allow for communication. That value is necessarily the same for all the microservices that belong to this pattern, it is thus a unique property that is shared by the microservices which in Listing 4.1 is split in multiple properties only as a consequence of the language limitation.

The absence of composite construction in the common metamodel is what makes the language structurally incapable of handling shared properties. Since every microservice has to be defined in only one place, it is completely distinct from the others.
Expressing interdependence requires a specific syntactical construct which is the proposition we will make in the next section.

```
version: '3'
services:
  registry:
    image: descartesresearch/teastore-registry
    expose:
      - "8080"
  db:
    image: descartesresearch/teastore-db
    expose:
      - "3306"
    ports:
      - "3306:3306"
  persistence:
    image: descartesresearch/teastore-persistence
    expose:
      - "8080"
    environment:
      HOST_NAME: "persistence"
      REGISTRY_HOST: "registry"
      DB_HOST: "db"
      DB_PORT: "3306"
  auth:
    image: descartesresearch/teastore-auth
    expose:
      - "8080"
    environment:
      HOST_NAME: "auth"
      REGISTRY_HOST: "registry"
  image:
    image: descartesresearch/teastore-image
    expose:
      - "8080"
    environment:
      HOST_NAME: "image"
      REGISTRY_HOST: "registry"
  recommender:
    image: descartesresearch/teastore-recommender
    expose:
      - "8080"
    environment:
      HOST_NAME: "recommender"
      REGISTRY_HOST: "registry"
  webui:
    image: descartesresearch/teastore-webui
    expose:
      - "8080"
    environment:
      HOST_NAME: "webui"
      REGISTRY_HOST: "registry"
    ports:
      - "8080:8080"
```

Listing 4.2: TeaStore's Docker Compose file with redundant properties grayed out

4.2 The Concern Concept

4.2.1 The need of a new model: Syntactical approach

Sharing a single property among multiple microservices requires an additional element in the metamodel for describing microservice architecture. As shown in Figure 4.4 the proposed model for an ADL allowing shared properties rely on a structure, named Concern here, that puts in relation sets of microservices with sets of properties. In that design microservices are defined partially in each Concern.

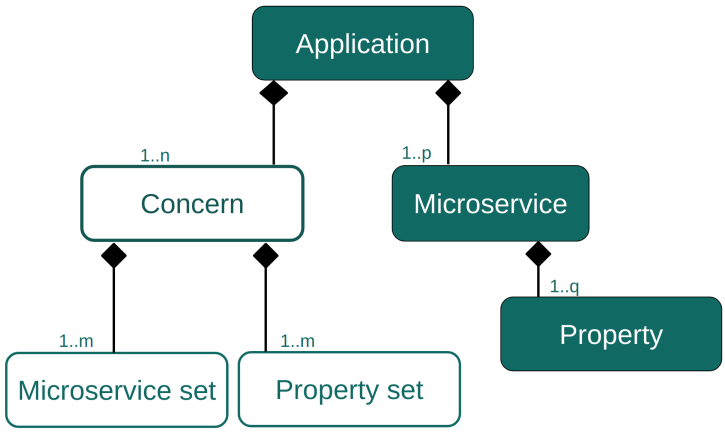


Figure 4.4: The concern-based metamodel for microservices ADLs.
In white the elements added to the common metamodel for microservices ADLs.

The new model is legacy-compatible as it offers the possibility to declare individual microservices without them being part of a concern. It is preferable as encapsulating properties of a concern that are not share with any other microservice would be superfluous and hinder writing as well as understandability. It also provides the users with the ability to use the concern concept at their convenience, not imposing the feature on them.

The two models are ultimately functionally equivalent. The Concern concept is abstract in the sense that it does not translate in an executable element, merging the different concerns produces a description matching the common model. As files describing microservice applications architecture are declarative the order of microservice and properties in the produced recomposition is inconsequential and thus any recomposition from the proposed model to the common model is equivalent.

Applied to the `REGISTRY_HOST: "registry"` property, that is part of the list environment attribute of services `persistence`, `auth`, `image`, `recommender` and `webui` in Listing 4.1, it produces an equivalent description as shown in Listing 4.3.

```
1 new Concern {
2   services{
3     [Set("persistence", "auth", "image", "recommender", "webui")]{
4       environment{ ["REGISTRY_HOST"] = "registry" }
5     }
6   }
7 }
```

Listing 4.3: Shared property expressed with the concern concept.

At *Line 1* a new Concern is declared. The `services` Concern attribute at *Line2* contains the relation between sets of microservices and sets of properties. The relation is materialized in the example as a map where the keys are sets of microservices and the values sets of properties. The map contains a single pair with a set of microservices declared at *Line 3* attached to a set containing a single property defined at *Line 4*.
The syntax of COMSA, the language we propose in this thesis, an therefore examples such as Listing 4.3, illustrating the Concern concept in a concrete syntax, are based on the Pkl syntax. Pkl [79] is a recent programming language developed by Apple aimed at configuration which blends aspects of Object Oriented Programming and functional programming.

Most of the aforementioned issues have been addressed. There is no more redundancy and the property is shared among multiple microservices in an evident manner that allows the reader to immediately understand the relation.
Modifying the value, would affect all the microservices located in the key part of the relation which makes the maintainability easy and much more resilient to error since it is not possible to overlook a microservice.

Architecturally the solution is still unsatisfactory as while it indicates correctly the link of multiple microservices to a number of properties, it does not express the structure of the application. Doing so requires to expand on the meaning of the Concern concept in a way that allows developers to not only capture the functional architecture of an application but also to express their intent and decisions when designing the system.

4.2.2 The adequate semantic approach: Concern

Definition of the Concept of Concern

While it is widely used in software development, the concept of "concern" does not have a clear definition. Tarr and Osser give a fitting description of that situation: "*'Concern' is a concept of which most software engineers have an intuitive grasp, but no precise definition. This is largely because there is such a variety of concerns of interest*" [96]. A concern is then something defined through its "*relevance to a particular stakeholder*" [47] and then, identifying that concern and which parts of a system are relevant is done, as Dijkstra

puts it,by "*focussing one's attention upon some aspect*". Once the aspect is identified the concern is "a predicate on units that indicates whether or not a given unit pertains to that concern"[96].

But if the concept of concern seems too large so that its particular instance could be encapsulated in a definition, [47] gives an interested perspective on the domain it covers as it considers ultimately a concern as a matter of relevance relatively to the "subject of an architecture description". Then a concern is essentially a particular view on an architecture.

To give an idea of the broadness of what can fall under that definition in a microservice context, we can first think of common matter such as logging, security, scalability, maintainability, monitoring or design patterns. But then any of these large domain of interest can contain more precise concerns. For instance, any design pattern such as "registry pattern" or "single database per microservice" is in itself a concern, a point of view on the system where the relevant elements are the microservices and their connections participating to this particular design pattern.

Another concern of a different nature from the other listed before could be particular naming conventions where the elements relevant to them are particular properties of microservices or connectors on which the naming convention is applied.

Those are aspects of the architecture in the sense that they express "the fundamental concepts or properties" as well as "the governing principles for the realization and evolution"[47] of the system.

The notion of concern was first explicitly formalized in the context of Aspect-Oriented Programming (AOP). AOP was introduced to address the limitations of dominant decomposition in programming languages by making cross-cutting concerns such as logging, security, or transaction management explicit and modularizable. In this context, a concern denotes a particular interest or responsibility that may span multiple program modules and cannot be cleanly encapsulated using traditional abstraction mechanisms [55, 54].

Crosscutting concerns

General definition A cross-cutting concern is a concern that affects several components of a system and intersects other concerns in some of them. The first phenomenon is called "scattering", the second "tangling"[95].

Theoretical situation A program can have multiple functions that read and write into files. We can then consider "reading" and "writing" as concerns of that program with the functions being the components of the system. Since multiple functions read and write into files, the concerns are scattered and since some functions do both of those actions they are also tangled together. In other words, the "read" concern crosscuts the "write" concern in some points and reciprocally. The "read" and "write" concerns are thus crosscutting concerns.

Scattering For a given concern, there are a number of components that participate in a given system. If that number is strictly superior to one, the concern is said to be scattered. The metric for quantifying the scattering of a concern or a system relies on proportionality between the number of components affected by a concern and the total number of components in a system[12].

Tangling For a given component, there are a number of concerns to which it is participating. If that number is strictly superior to one, the component is said to be tangled. The metric for quantifying the tangling of a component or a system relies on proportionality between the number of concerns affecting the component and the total number of concerns[12].

Dominant decomposition Languages used to describe (or implement) a system is the main determinant regarding the dominant concern around which the system is decomposed. A language provides fundamental concepts that allow for certain - usually one - ways of modularizing the system around the dominant concern. That has been called the "tyranny of the dominant decomposition" [95] and has as a consequence to relegate others concerns to sub-descriptions, thus producing scattering among and tangling in the components. For instance, the description of a program through a language whose concepts allow for data modularization would relegate code elements such as functions affecting data structures at an inferior level of relevance producing their scattering among described data structures. Reciprocally in a procedural programming language, references to data structures are scattered among and tangled in procedural structures.

In the context of microservice architectures, architectural patterns such as service discovery, dedicated databases, API gateways, or externalized configuration naturally qualify as *architectural cross-cutting concerns*. These patterns define constraints, shared properties, and interaction rules that apply to multiple microservices simultaneously, independently of their business responsibilities. As such, they are scattered across microservice descriptions and tangled with other architectural and technical concerns when expressed using microservice-centric languages. Policies such as restart, logging or monitoring policy qualify as *architectural cross-cutting concerns* for the same reasons.

Concerns applied to Microservices Applications

Microservices applications have specific cross-concerns. Some of them are largely immaterial such as team ownership of microservices and are not identifiable in code and description files. Others are scattered among the microservice description and require analysis to be exposed. The following Docker Compose code used to deploy the well known TeaStore microservice application [56][20] is a good example of that phenomenon.

4.2.3 Concerns Identification in the Teastore Application

The Teastore application provides a representative illustration of how classical microservice architectural patterns manifest as cross-cutting concerns. Each identified pattern corresponds to a distinct architectural decision whose realization spans several microservices and whose expression is fragmented across the description file. The reason for the high level of scattering in the description of microservices' applications is a consequence of the widely shared microservice application base metamodel where "each Application owns a none-empty set of microservices. Each Microservice has a unique distinct identity msid (e.g. registry) and owns a non-empty set of properties. Each Property has a unique distinct name (e.g. image) and a value (e.g. descartesresearch/teastore-registry)"[70]. The system's unit of decomposition being the microservices, each cross-cutting concern is scattered among the microservices description blocks. In mirror of that, microservices' description blocks are subjects of a high level of tangling.

Examples of Concerns in Microservice architecture Listing 4.4 shows a variety of cross-cutting concerns that are identifiable from the analysis of the properties of the system. Others could be and are probably existent in the system but are not detectable in the system description. Some cross-cutting concerns could be specific to a particular application while others are frequently found in applications. Examples would be logging, monitoring, security, fault-tolerance, naming conventions or design patterns.
Each of those concerns are rather large and can be developed into more specific concerns. Design patterns, i.e. identified architectural solution, for instance are applied over all areas of microservice application development. Because of that they appear as a good starting point for cross-cutting concern identification.

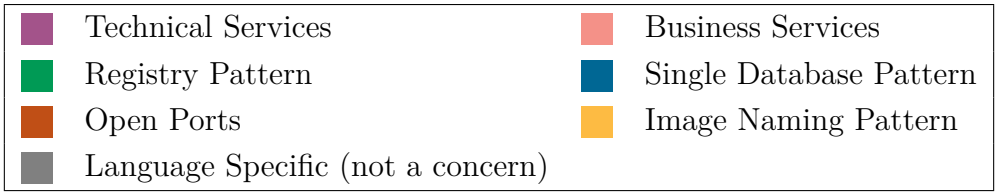
The concern concept versatility makes it adequate to describe a microservice application. The different points of view that can be taken are of different natures. For instance, a naming template for a property can be as relevant to identify as a communicational pattern.
Also, any single system can be described in a number of different manners. Pushed to the extreme, each property can by itself be seen as a particular concern. The process of grouping them to form meaningful concerns is not self evident and thus reconstructing the intended architecture can prove a challenge.
In addition key informations could be hidden in files that are not necessarily accessible to the reader and even when they are, looking for the relevant architectural informations is a tedious and time consuming process. In Listing 4.1 only the name of the services and the `image` attribute is necessarily expressed in the file as all the others could be located in the services Dockerfile. The reader can then only rely on the arbitrary names decided by the developers that can make sense to them without that being the case for another person. That's particularly obvious when considering the technology used for a microservice which is a fairly natural and common choice for a microservice identifier. A specialised

technology can be analogous to another relying on another, the name would differ and the architectural function be lost for a reader lacking the appropriate knowledge.

An example of obscure role for a microservice is the `persistence` microservice in Listing 4.1 at *Line 13*.
The `persistence` service acts as an intermediary between the database and the microservices using it. The name that was chosen for it comes from a specific Java technology named Jakarta Persistence whose main function is to be an Object Relational Mapper, i.e., making the connection between objects in Object Oriented Programming and corresponding data structures in relational databases.
The way this technology is used in the Teastore application is uncommon as the Jakarta Persistence library is usually directly integrated into the Java code of the business microservices using the database and not made an independent microservice. However it is consistent with the microservice style as it provides better options in terms of future technological freedom of choice than it being embedded in business code.
A first hindrance to architectural understanding is the choice of he name of the technology for identifier instead of the architectural role of the microservice. In the same way, the `db` microservice has an identifier that describes its function, distinguished from the particular technology used to instanciate it, `persistence` has at best an unclear name.
A second problem lies in the properties only indicating the connection between the `persistence` and `db` microservices. The `DB_HOST: "db"` property at *Line 20* of `persistence` points to the `db` service name at `Line 7` and the `DB_PORT: "3306"` property at *Line 21* points to the `expose: - "3306"` property at *Lines 9 and 10* of the `db` service.
The exclusive relation between the microservices `image` and `recommender` to `persistence` cannot be deduced from the properties and without a specific structure in the language to describe it, parallel documents are needed such as the diagram in Figure 4.2.

```yaml
version: '3'
services:
  registry:
    image: descartesresearch/teastore-registry
    expose:
      -"8080"
  db:
    image: descartesresearch/teastore-db
    expose:
      - "3306"
    ports:
      - "3306:3306"
  persistence:
    image: descartesresearch/teastore-persistence
    expose:
      -"8080"
    environment:
      HOST_NAME: "persistence"
      REGISTRY_HOST: "registry"
      DB_HOST: "db"
      DB_PORT: "3306"
  auth:
    image: descartesresearch/teastore-auth
    expose:
      -"8080"
    environment:
      HOST_NAME: "auth"
      REGISTRY_HOST: "registry"
  image:
    image: descartesresearch/teastore-image
    expose:
      -"8080"
    environment:
      HOST_NAME: "image"
      REGISTRY_HOST: "registry"
  recommender:
    image: descartesresearch/teastore-recommender
    expose:
      -"8080"
    environment:
      HOST_NAME: "recommender"
      REGISTRY_HOST: "registry"
  webui:
    image: descartesresearch/teastore-webui
    expose:
      -"8080"
    environment:
      HOST_NAME: "webui"
      REGISTRY_HOST: "registry"
    ports:
      -"8080:8080"
```

Listing 4.4: TeaStore's Concerns highlighted in the Compose file



Such uncertainties make difficult after the fact concern attribution, but reasonable, and most importantly, correct decisions can be made based on static analysis and sometimes with the help of additional informations.
Listing 4.4 shows concern division of the file in Listing 4.1 file through coloring the different properties.

Distinguishing technical from business services is central in architectural description although it is not an always a clear-cut choice since technical microservices often include some business aspects and vice versa. The choice to consider `registry` and `db` as technical services whereas the others are considered business services comes from the fact that they are to only two to which related properties can be identified outside their own description.

Service registry pattern is an architectural cross-cutting concern that can be identified through multiple properties scattered among microservices in the Teastore application. In the `environment` attribute of `persistence`, `auth`, `image`, `recommender` and `webui` we can identify the central properties being part of service registry pattern.
`HOST_NAME :` associated with the service identifier *(Lines 18, 27, 34, 41 and 48 of Listing 4.4)* is designed to transmit the microservice identifier to the `registry` service. It is to be noted that here again there is an inconsistency as the service identifier is hard-coded and is at risk of being erroneous because a typographical error. The better option would be to indicate that the value is the service identifier but YAML, the concrete syntax used for Compose in this example offers no means to do so.
`REGISTRY_HOST: "registry"` *(Lines 18, 27, 34, 41 and 48 of Listing 4.4)* is the identifier of the registry service being hard-coded in the environment variables of microservices belonging to the pattern. It is used as a means of connection to the registry service and ultimately to the other connected microservices. The remark made about the `HOST_NAME` property is also relevant here as regarding consistency, error avoidance and maintainability it would be preferable to indicate the relation between the identifier and the environment variable value.
The `expose: -"8080"` property is not as obviously part of the service registry pattern. However it indicates a point of connection shared by all the microservices belonging to the pattern that permits communication which justifies the decision of including it to the concern.
Aside from pointing out the properties related to the service registry pattern in Listing 4.4 offers by contrast an important insight which is that the `db` service is not part of it.

A dedicated database is a database that is used by a single microservice. It follows a microservice architectural pattern named "Single Database per Service" [68]. It is considered a good practice as it keeps microservices independent [73] [77] but, while largely adopted, its usage implies a tradeoff between indepedence and data consistency

[60].
Through static analysis we, can identify that `db` is only connected to `persistence`. `db` exposes a different port to the application network than the other microservices signified through the property `exposes: -"3306"`. `persistence` is the only microservice connecting to it through the properties `DB_HOST: "db"` and `DB_PORT: "3306"` contained in its `environment` attribute.

Entry points identification is not a structural pattern but a accessibility and security concern of importance.
Having the open ports scattered among the application is a risk of overlooking and thus risking giving a mean of entry to a malevolent actor. This is the case here with the properties `ports: - "3306:3306"` being tangled with other properties of `db` at *Lines 11 and 12* and `ports: - "8080:8080"` at the very end of the file at *(Lines 50 and 51)* in `webui` description.

Naming patterns don't have an effect on the behavior of the application. However they impact maintainability and understanding by systematizing how to write and analyse the description. To the difference of property values contrained by their intrinsic signification, the naming patterns apply to properties whose values are determined through the judiciousness of the developer. The ability to factorize is also one of the pros of naming patterns.
In Listing 4.4, we can identify a consistent naming pattern as the `image` attribute at *Lines 4, 8, 14, 23, 30, 37 and 44* has a value of `descartesresearch/teastore-` concatenated with the service identifier. Unfortunately, the Compose language as it follows the common metamodel is unable to to represent this commonality between the different values that can be considered as a shared value that is actually a function parametrised by the service identifier. A consequence of that inability is populating the description file with redundancy, hindering understandability, maintainability and increasing the risk of error. Electing the image naming pattern as a structuring concern is an example of a choice made in the presentation of the application. In Listing 4.4, the `image` property of `registry` and `db` could have easily been integrated into the concerns representing the architectural patterns they are at the center of.

Scattering of the identified concerns is significant. The Single Database and Open Ports concerns are the least spread, each of them reaching 2 out of 7 microservices. On the other end, the Image Naming concerns is maximally spread out with all seven microservices pertaining to it. The Registry Pattern concern contains six out of seven microservice leaving only the `db` microservice being not part of it.

Important missing concerns are to be considered. The web frontend concern observable in the diagram in Figure 4.2 is of importance as it indicates the way the application is

meant to be used and the communicational structure of the application from a business point of view. Also the limited sharing of `persistence` is not expressible through the properties but is significant to the understanding of how the persistent data is managed.

This perspective shows how the Concern abstraction which is the cornerstone of COMSA as the fundamental first-class architectural construct specifically designed to capture microservice architectural patterns and policies as explicit, modular, and reusable cross-cutting concerns.

4.3 Concrete Syntax

4.3.1 Fundamental properties of the Language and implementation needs

The implementation of COMSA relies on a limited set of necessary concepts that define the scope and expressive power of the language. The `Concern` is the main addition in terms of first-class architectural element, as it captures architectural decisions that span multiple microservices. Microservices themselves remain first-class citizens of the language, primarily to ensure retrocompatibility with the common microservice ADL model and to allow partial or concern-independent service descriptions.

The model also requires explicit notions of collections, in particular lists and sets, as concerns are defined through relations between sets of microservices and sets of properties. In addition, the ability to assign functions to properties, at least parameterized by the microservice identifier, is necessary to express a large class of concerns in a concise and semantically meaningful way, such as naming patterns or derived configuration values.

Architectural constraints are at time of writing tied to Concerns definition. A few are already implemented, adding others would require modifications to the COMSA module and are not directly expressible in description files. Constraint verification is delegated to dedicated analysis tool described in Appendices B.2.1 B.2.2. Finally, the language allows the definition of behavioral rules associated with architectural patterns, enabling the derivation of implicit properties from the use of design patterns expressed as concerns.

4.3.2 Rationale for the choice of Pkl

The concrete syntax of COMSA is based on the Pkl language[79]. The choice of Pkl is motivated by its positioning as a configuration language that combines a declarative nature with the expressiveness of a general-purpose programming language. Pkl has been designed specifically to define configuration formats while providing an interpreter capable of evaluating non-literal values, functions, and compositions, which makes it particularly suited for architecture description. This combination makes it possible to define concise

and readable architecture description files, usable as configuration files, while retaining strong abstraction and reuse mechanisms. In particular, the object-oriented aspects of Pkl provide a structurally sound basis to implement extensible metamodels. COMSA is implemented as an extension to the Pkl language. The inheritance mechanism allows COMSA to introduce the `Concern` concept as a new architectural abstraction while reusing and extending existing structures. This design choice makes it possible to define libraries of reusable concerns in a natural and type-consistent manner. The Compose model has been chosen as the basis for describing microservice properties. Although Compose is commonly associated with container-based deployment, its service description model provides a large set of properties that are largely technology-agnostic and correspond to widely shared microservice concepts. This makes it a suitable intermediate representation, facilitating compilation toward existing execution technologies while remaining sufficiently abstract to allow for any compilation target.

4.3.3 Teastore expressed with the concern concept

Reorganising the file through concerns in accordance to the concern model illustrated in Figure 4.4 removes both scattering and tangling and produces benefits in terms of understanding, coherence, maintanability and error avoidance. That can be seen throught the Teastore example, rewriting the application description using the concern concept and the concerns we have identified in Listing 4.4.

```
1  // Registry global settings
2  const REGISTRY = "registry"
3  const REGISTRY_PORT = "8080"
4  // Database global settings
5  const DB = "db"
6  const DB_PORT = "3306"
7
8  const BusinessServices = Set("persistence", "auth", "image", "recommender", "
       webui")
9  const TechnicalServices = Set(REGISTRY, DB)
10 const AllServices = BusinessServices + TechnicalServices
11
12 function publish(port: String): String = "\(port):\(port)"
13
14 version = "3"
```

Listing 4.5: Teastore expressed in concerns (part1)

Global variables and significant sets of microservices can be isolated from the rest of the file as shown in Listing 4.5.
The concrete syntax is based on the **Pkl** syntax upon which a dedicated module has been developed. The keywords that belong to **Pkl** are highlighted with the same `color`.
Through *Lines 1 to 6* of Listing 4.5, two kinds of variable are defined. At *Lines 3 and 6*, ports values are encapsulated into variables. More interestingly, at *Lines 4 and 7*, the

identifiers of respectively `registry` and `db` are encapsulated into variables. An interesting opportunity given by Pkl is that those identifiers will be usable to define properties of those services as well as in other services properties to refer to them.
Sets of services can be defined to identify relevant groups. In accordance to the useful distinction between business and technical services, corresponding sets of identifiers are established through constants `BusinessServices` at *Line 8* and `TechnicalServices` at *Line 9*, the latter containing the constants to which `registry` and `db` have been assigned. As this division is exhaustive relatively to the services constituting an application, their union is used to set the `AllServices` variable at *Line 12* for clarity.
Finally, using the possibilities offered by Pkl, the underlying descriptive language that is also a programming language, a function named `publish` is declared at *Line 14* only for easiness of writing, avoiding repetition and in some way clarifying the effect of the `port` compose attribute.

```
15 concerns {
16   // Technical Patterns
17   ["Service Registry Pattern"]
18   = new Concern{
19     services {
20       [BusinessServices] {
21         environment {
22           ["REGISTRY_HOST"] = REGISTRY
23           ["HOST_NAME"] = module.SID
24         }
25       }
26       [BusinessServices.add(REGISTRY)] {
27         expose { REGISTRY_PORT }
28       }
29     }
30   }
```

Listing 4.6: Teastore expressed in concerns (part2)

The registry pattern is expressed in Listing 4.6 using only the concern concept.
The beginning of the application description is signaled at *Line 15* with the `concerns` keyword that is specific to the COMSA module. It is a `Mapping`, a data structure specific to Pkl analoguous to a map, i.e., a set of key-value couples, and the core of the concern-based application description.
In Listing 4.6, `["Service Registry Pattern"]` at *Line 17* is the identifier of the Registry pattern with the concern object description being located between *Lines 18 and 30*. The Pkl keyword `new` is used to declare the `Concern` object at *Line 18*. The main section of a concern describes a relation between sets of services and sets of properties assigned to them. That relation is located in the `services` attribute of the `Concern` object beginning at *Line 19* in the form of a `Mapping` where keys are sets of microservices and values are partial Compose services whose properties are attributed to all of the microservices of the key set.

The first services-properties relations is located between *Lines 20 and 25*. At *Line 20*, the designated set of microservices is `BusinessServices` that has previously been defined at *Line 8* of Listing 4.5. The `environment` Compose service attribute is modified for that all the services in the service set with two properties added to it. At *Line 22*, `["REGISTRY_HOST"] = REGISTRY` combines the property scattered among five microservices in Listing 4.1 making it effectively one property. The environment property is set with the `REGISTRY` variable defined at *Line 2* of Listing 4.5.
At *Line 23* the `["HOST_NAME"] = module.SID` sets the environment variable of each of the microservices in `BusinessServices` with its own identifier in the application description file. To the difference of the previous property, it does not factorize multiple identical properties but expresses an implicit relation in Listing 4.1.
Those two properties are the heart of the Service Registry Pattern as they permit the communication between microservices to happen. However as the `expose` attribute documents the internal communication port for all microservices in the application we made the choice of including in the Service Registry Pattern with it being set at *Line 27*. As this property is also shared by the registry microservice, it is added to the associated microservice set at *Line 26* with using the Pkl `add` set method. In accordance with the functional and declarative nature of pkl, `BusinessServices` is not modified, instead a new set containing the elements of `BusinessServices` and the value of the `REGISTRY` variable, i.e., the registry identifier, is returned.
In summary, the Registry Pattern expressed through a concern is a prime example of how the concern concept permits architectural clarity and that while adding new language entities it actually diminishes the number of properties in a description. This description is nonetheless equivalent but more coherent, and has removed all scattering of the concern in the file making it more easily maintainable and less error prone.

```
31  ["Database Per Service Pattern"]
32  = new Concern {
33    services {
34      [DB] {
35        expose {DB_PORT}
36      }
37      ["persistence"] {
38        environment {
39          ["DB_HOST"] = DB
40          ["DB_PORT"] = DB_PORT
41        }
42      }
43    }
44  }
```

Listing 4.7: Teastore expressed in concerns (part 3)

The Dedicated Database pattern presented in Listing 4.7 is the second concern expressed in the file. To the difference of the Service Registry pattern in Listing 4.6, it

does not factorize identical properties of Listing 4.1 but serves only for clarity in the description of the architecture. For ease of usage, it is possible to use single microservices identifiers as key in the mapping as shown in *Lines 34 and 37*.
This concern encapsulates the database pattern with clarity and removes the scattering that clouded its delimitations an pertaining properties in Listing 4.1.

```
45  // Naming Patterns
46  ["Image Template"]
47  = new Concern {
48    services {
49      [AllServices] { image = "descartesresearch/teastore-" + module.SID }
50    }
51  }
```

Listing 4.8: Teastore expressed in concerns (part 4)

The image naming pattern that is scattered among all microservices in Listing 4.4 is captured in a single concern and property wise in the sole *Line 49* of Listing 4.8.
The total commonality of the property is expressed through the `AllService` identifier defined for ease and clarity in Listing 4.5. The associated property set contains only the image template expressed as a function whose return value is the concatenation of a shared prefix and the service identifier.

```
52  // Security
53  ["Public Ports"]
54  = new Concern {
55    services {
56      ["db"] {
57        ports {
58          publish(DB_PORT)
59        }
60      }
61      ["webui"] {
62        ports {
63          publish(REGISTRY_PORT)
64        }
65      }
66    }
67  }
```

Listing 4.9: Teastore expressed in concerns (part 5)

Identifying publicly exposed ports can be seen as the most obvious concerns when thinking about security. The `ports` attribute that indicates the entry points is nonetheless mixed up with all other properties in Listing 4.1. In particular the more frequent, less significant and closely looking `expose` attribute makes it difficult to spot all of the open ports in the application.
The `"Public Ports"` concern presented in Listing 4.9 emphasize the entry points by

groupings relevant services and ports in the same code block.

```
68  // Patterns without related properties
69  ["Web Frontend"]
70  = new Concern {
71    services {
72      ["webui"]{}
73      [Set("persistence", "auth", "image", "recommender")] {}
74    }
75  }
76  ["Shared Service"]
77  = new Concern {
78    services {
79      ["persistence"]{}
80      [Set("image", "recommender", "auth")]{}
81    }
82  }
```

Listing 4.10: Teastore expressed in concerns (part 6)

Invisible architectural patterns are those have no related properties in the application description. Figure 4.2, which is documentation provided by the application developers, indicates two sets of connections between microservices that are not detectable through analysis of Listing 4.1.
The first one is the relation between the frontend of the application, `webui` and the other business services. The second one is the sharing of the `persistence` service between `image`, `recommender` and `auth` that ultimately indicates the use of the database.
The Concern concept allows for expression of patterns empty of all properties. In that case, it only serves as architectural information and permits to not rely on external documentation.

Concern based expression of the Teastore application has entirely removed property scattering and service tangling. The factoring of identical properties made the number of properties needed to describe the architecture less than a third of what it was in the original file, going from 28 to 9. The concern based description remains equivalent to the one following the common model and does not factorize with the intent of being economical in its expression but to increase coherence and highlight the patterns lying in the application. Through the usage of the `Concern`, we have shown that it is possible to express any kind of concerns as long as it materialize in the form of sets of microservices linked to sets of properties which are the substance of any description.
However that unconstrained field of expression is limited in terms of standardisation and thus makes the identification of concerns difficult though multiple applications. To that effect it is desirable to establish a library of reusable concerns to provide architectural guidelines and standardize the descriptions of microservices applications to which we dedicate the next chapter.

4.3.4 Implementation: File Structure and Concern Class

The concrete syntax of the COMSA model presented in this thesis is encoded with the Pkl language, which provides a declarative format for the end user as done with the teastore application in Section 4.3.3. The declared application architecture is then processed using the structures declared in the COMSA module through the `Pkl` program which definitely determines the non litteral values of variables. Ultimately the resolved system can be rendered in the Pkl syntax or converted to other formats such as YAML or JSON.

```
1 open module comsa
2 extends "./compose.pkl"
3 //...
4 concerns: Mapping<String, Concern>?
```

Listing 4.11: `concerns` *module property* in the `comsa` module (simplified)

The `comsa` module is actually an extension of the `compose` module as shown at *Line 2* of Listing 4.11. The motivation for choosing the compose representation of microservices is that even though it assumes an underlying container technology, most of the attributes are not specific to it. They appear as a limited extension of a general concept of independent processes whose inputs and outputs are connected to a network which is the mainstream characterization of microservices and the one we have adopted.

The structure of comsa files is described at *Line 4*, declaring a *module property* named `concerns` that is a `Mapping` of `Strings` to `Concern`. The question mark sign, `?`, indicates that `concerns` can be null, which is equivalent to being empty. As the COMSA model is an extension of the common model (see Listing 4.1) of microservice ADLs, and in the present concrete syntax of the Compose language, microservices can be declared outside of concerns. The *module property* `services` shown in Listing 4.12 imported from the `compose` module can be used to that effect.

```
1 services: Mapping<String, Service>?
```

Listing 4.12: `services` *module property* in the `compose` module (simplified)

The object oriented aspect of Pkl provides the ability to declare classes as illustrated in Listing 4.13 with the Concern class. The core of the Concern class lies in the *class property* `services`. It captures the relations between `Collections` (either `Set` or `List`) of services Identifiers (`Id`), that are `Strings` constrained by a regular expression, and a `Service` that act as a partial description of the associated microservices. For ease of use the `Mapping` keys can also be a single `Id` but are ultimately treated as a `Collection`

containing a single element.

```
open class Concern {
  services: Mapping<Id|Collection<Id>, Service>?
  //...
}
```

Listing 4.13: `Concern` class in `comsa` module (simplified)

The definition similarity between the `Concern` *class property* `services` and the `compose` *module property* of the same name is only syntactic. While they both associates `Service` objects, that as shown in Listing 4.14 are a Pkl implementation of Compose services, to service identifiers, the semantical implications are very different. On one hand, the Compose association between identifier and properties implies, respectfully to the common model, and functionally by using a single `Mapping`, that each service is defined exhaustively in a single place, with very limited contingent exceptions. On the other hand, the `Concern` *class property* spreads the definitions in the different instances of the `Concern` class, allowing for a distributed definition that are meant to be unified at compile time.

```

open class Service {
  //...
  image: String?
  //...
  environment: ListOrDict?
  //...
  ports: Listing<Number|String>?
  //...
}
services: Mapping<String, Service>?
```

Listing 4.14: `Service` class in the `compose` module (simplified)

Chapter 5

Library of Concerns for microservice applications

Having presented the Concern concept and construction as the basis of the COMSA language, we dedicate this chapter to the establishment of a library of specific reusable Concerns. Through them we provide ready to use architectural types making the applications architecture explicit by design. We also introduce a fundamental distinction between property centric concerns and service centric concerns. The later type allows us to simplify the applications descriptions as some properties can be deduced from the use of particular concerns and can be therefore made implicit.

5.1 The need for a concern library

We have established through the Teastore application example that the use of the concern concept can greatly reduce the proliferation of properties in a file by of having the system *"dominant decomposition"* [95] on concerns instead of microservices .
Through that approach property instances are reduced to their minimum, that is, one. Also the unification of different properties around a designated common concern not only increases the architectural structure of the system description but actually provides a lacking one by focusing *"on the high-level structure of the overall application rather than the implementation details of any specific source module"* [**medvidovic2000**] which was the state of microservices ADLs. Microservices and properties are not discarded but subordonnated in the description to the architecture of the system.
From an architect point of view, the concern concept allows to express its vision of the application without being constrained by the predefined categories of the language. That freedom permits to have application-specific point of views by not being limited by built-in structures of the language.

However the absence of specific predefined reusable concerns makes it harder to express the commonly found concerns that are, by definition, the basis of most of existing

applications.
For instance, Listing 4.6 instanciates the Service Registry pattern that, while not being an absolute necessity for a microservice application, is so central to the majority of them that it is exists de facto in Docker-hosted application through Docker Networks[24] allowing microservices to communicate seemlessly using only microservices identifiers.
While the patterns are well captured in Listing 4.5 to Listing 4.10, their identification to a reader depends greatly on the naming choice of the developer since they are all `Concern` objects.

Microservices roles in concerns are also not made explicit by design. In Listing 4.7, the distinction between the database and the client service is made clear contingently by the custom names decided by the developer, which is a gamble relatively to the reader understanding and an unnecessary workload.

Finally properties essential to a pattern could be made implicit with specified patterns, which would at a minimum reduce the amount of properties in a file or inject missing but desirable properties. For instance in Listing 4.10, the gateway pattern does not have any properties. However we understand that in a deployment the public entrypoint of the application is meant to be deployed last to not provide an unstable experience to the user. Such properties could be automatically derived from a specific pattern.

5.2 Revisiting the Teastore Example

Listing 5.1 modifies the `concerns` section given in Section 4.3.3 by using specified concern classes. While the description is largely similar to the previous version notable differences can be spotted.

Specialized `Concerns` objects are declared for each identified concern (*Lines 3, 19, 33, 38, 45, 50*), which makes the identification of the concerns nature not dependant on the concerns identifiers. Among the concerns, we can differentiate two sets:

- **Property Centric Concerns:** The concerns `ImageConcern` (*Line 33*) and `PublicPortsConcerns` (*Line 38*) define the values of a service property in a centralized manner. That is particularly true of the `PublicPortsConcerns` that merely act as a gatherer of identifier-value associations to highlight a particularly critical aspect of the application, here being the entry points of the application for external agents.
  The `ImageConcern` however contains more than a signal for attention but captures the common repository and naming pattern of all the services of the application in their image form. To that effect it possesses the `image` class specific related property that can either contain a literal value or, as it it the case here, a function parametrized by the service identifier.

The `sids` class property is conceptually more impactful as it traverses most `Concern` derived classes. It captures the identifiers of services to which the properties are to be affected to, conveniently distinguishing the two for clarity.

- **Service Centric Concerns:** The patterns `ServiceRegistryPattern` (*Line 3*), `DatabasePerServicePattern` (*Line 19*), `ApiGatewayPattern` (*Line 45*) and `SharedPattern` (*Line 50*) are structured around a microservice that gives its name to the pattern and determines its function.

The `msid` class property at *Lines 4, 20, 46 and 51* is destined to receive the identifier of the central service and distinguish it from the services encompassed in the pattern, which identifiers are contained in the `sids` attribute.
The `msid` and `sids` *class properties* are associated to the optional `service_template` (*Lines 6 and 25*) and `main_template` (*Line 22*) class properties that are Pkl Compose `Service` objects acting as templates for partial definition of the services.

Since all of those classes derive directly or indirectly from the `Concern` class, supplementary properties that appear to be attachable to the concern can still be added to the declaration through the `services` class property (*Lines 12-15*).

```
concerns {
  ["Registry"]
    = new ServiceRegistryPattern {
        msid = "registry"
        sids = BusinessServices
        service_template {
          environment {
            ["REGISTRY_HOST"] = msid
            ["HOST_NAME"] = module.SID
          }
        }
        services {
          [BusinessServices + Set("registry")] {
            expose { REGISTRY_PORT }
          }
        }
      }
  ["Database Per Service Pattern"]
    = new DatabasePerServicePattern {
        msid = "db"
        sid = "persistence"
        main_service_template {
          expose { DB_PORT }
        }
        service_template {
          environment {
            ["DB_HOST"] = msid
            ["DB_PORT"] = DB_PORT
          }
        }
      }
  ["DescartesresearchTeastore Images"]
    = new ImageConcern {
        sids = BusinessServices + Set("registry", "db")
        image = "descartesresearch/teastore" + module.SID
    }
  ["Public Ports"]
    = new PublicPortsConcern {
        ports {
          ["db"] { publish(DB_PORT) }
          ["webui"] { publish(REGISTRY_PORT) }
        }
      }
  ["WebUI"]
    = new ApiGatewayPattern {
        msid = "webui"
        sids = Set("persistence", "auth", "image", "recommender")
      }
  ["Persistence"]
    = new SharedService {
        msid = "persistence"
        sids = Set("image", "recommender", "auth")
      }
}
```

Listing 5.1: Teastore with concern library

Using the `topology` tool described in Section B.3.1 we can produce the graph in Figure 5.1 using only its COMSA descrtion presented in Listing 5.1. The topology of the application is represented with the same graph by the Teastore developers and shown in

Figure 4.2 adding architecture information but distinguishing patterns and indicating open ports. The more important point is that it is entirely contained in the COMSA description showing the ability of the language to accurately describe the intended architecture. Through usage of a concern-based language we then have a single reference for deployment and architecture description, text-based or graphical leaving no room for inconsistencies.

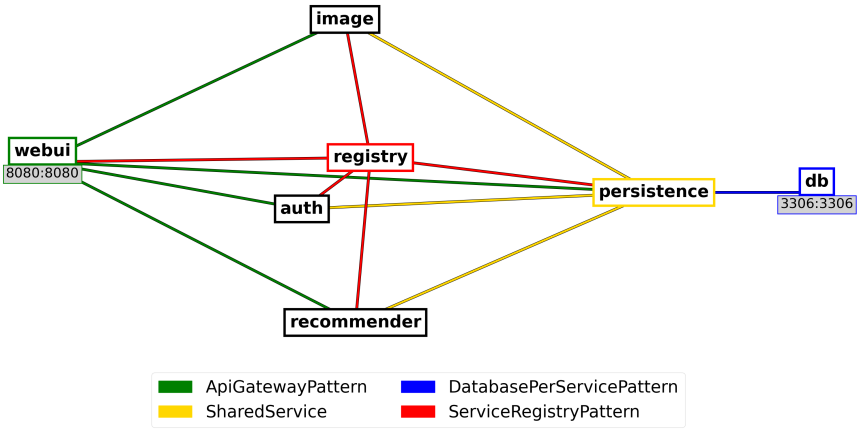


Figure 5.1: Topology of the Teastore application generated from its COMSA description.

5.3 Concern Library Architecture

In accordance to the reasons given in Section 5.1, we have established a Concern library. As shown in Figure 5.2, all classes derive from the `Concern` class.

The `ClusterConcern` and `ServiceCentricConcern` classes, along with their derived classes, serve as general purpose classes representing patterns that are either not implemented in the library or specific to a particular application. A `ClusterConcern` indicates a group of microservices whose grouping is relevant in representing the application online mechanism. The `ServiceCentricConcern` has the same signification but adds a distinction between a particular service, central to the pattern, and the others that belong to the pattern by their connection to the central service.

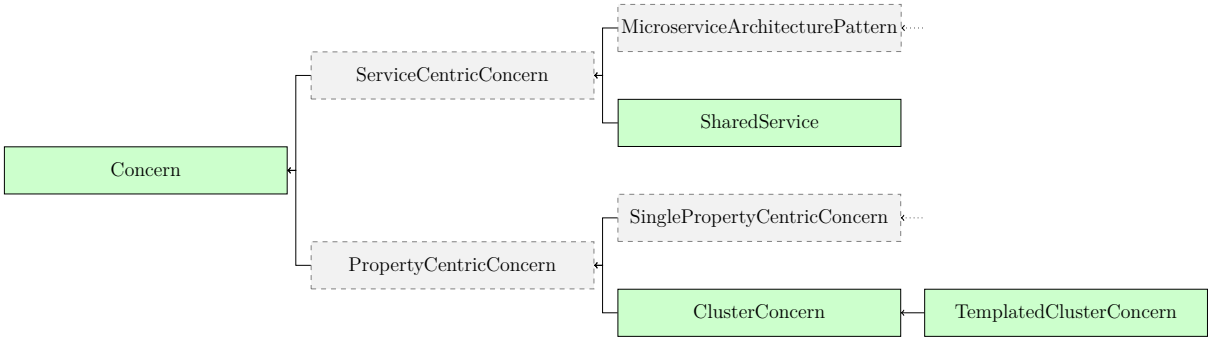


Figure 5.2: Classes derived from the `Concern` class.
Concrete classes are green, abstract classes are grey.

5.3.1 `PropertyCentricConcern`

Properties shared among a large number of services are frequent in application descriptions. It usually represent, an architectural decision that can be of different nature such as a naming pattern, a common attribution, or system-wide policy. Those concerns are highlighted and factorized by `Concern` objects whose type derived from the `PropertyCentricConcern` class that are graphically presented in Figure 5.3. Property centric concerns are of the smallest granularity for a concern as they define only one property for a set of services. To the difference of service centric concerns, their architectural relevance is not guaranteed by the simple fact that they are usable or even have an important factorizing effect. To be used appropriately, they must capture an architectural decision that is fully contained in a single property.

The `sids` class property introduced by the `PropertyCentricConcern` class, is central to all the property centric classes as it designates the services to which the property is assigned. In relation to the `sids` property, the property centric concerns possess a property to set the value defining the concern. For instance, the `ImageConcern` that is used to define the value of the `image` attribute of designated services has a mandatory class property of the same name as shown in Listing 5.2.

```
["image_concern_example"]
  = new ImageConcern {
      sids = Set("service_1", "service_2")
      image = "image_location"
  }
```

Listing 5.2: ImageConcern example

Functions parametrized by the service identifier can be used to define any property using `module.SID`. They are particularly useful for property centric concerns as Listing 5.3 shows.

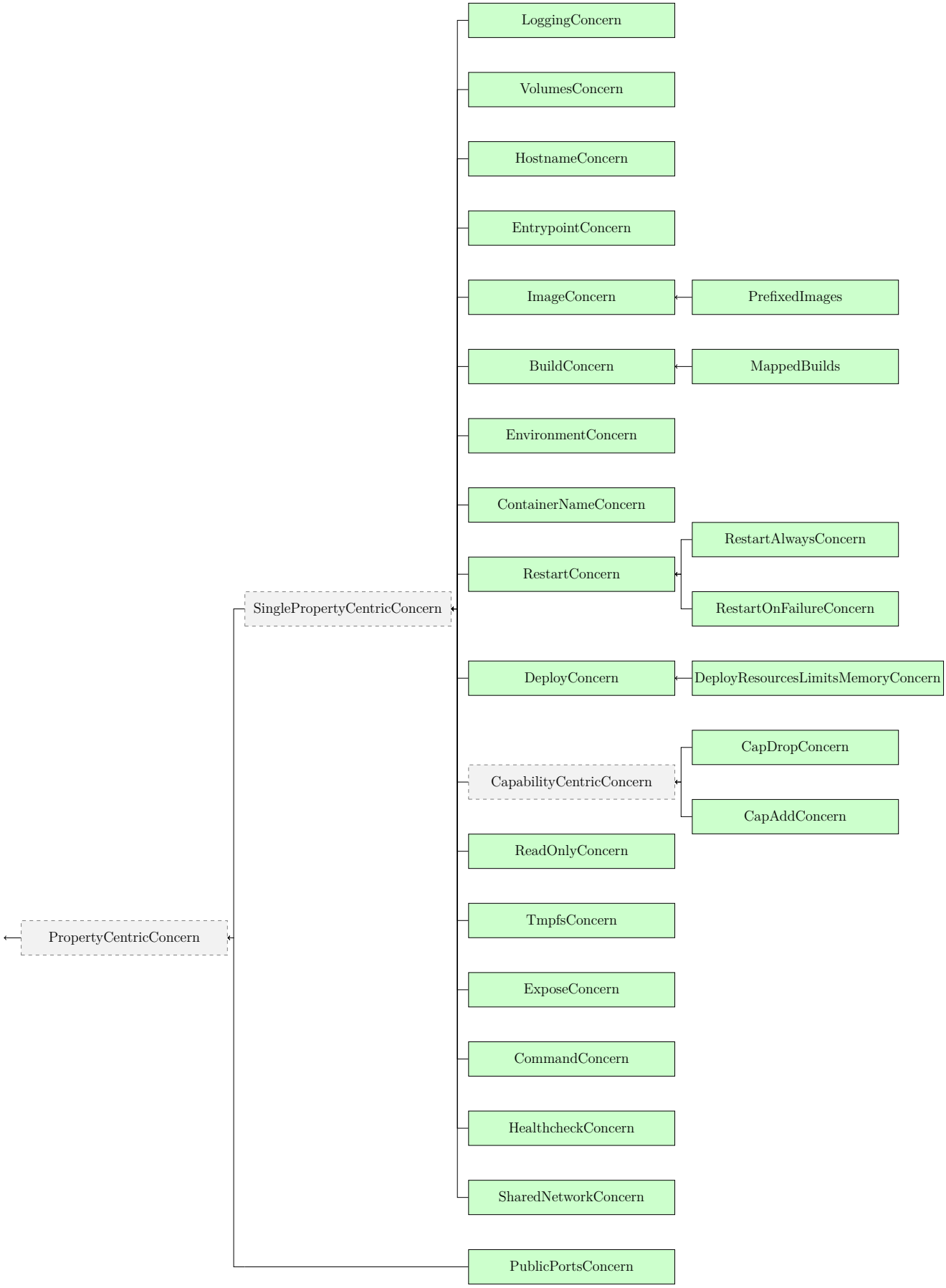


Figure 5.3: Classes derived from the `PropertyCentricConcern` class.
Concrete classes are green, abstract classes are grey.

```
["image_concern_example"]
  = new ImageConcern {
      sids = Set("service_1", "service_2")
      image = "image_repository/" + module.SID
  }
```

Listing 5.3: ImageConcern example with function

5.3.2 `MicroserviceArchitecturePattern`

Identified microservice patterns are by definition insightful and practical. In COMSA, they are derived from the `MicroserviceArchitecturePattern` class and grouped around intermediate classes distinguishing large area of interest. Figure 5.4 gives an overview of the patterns implemented in the library.

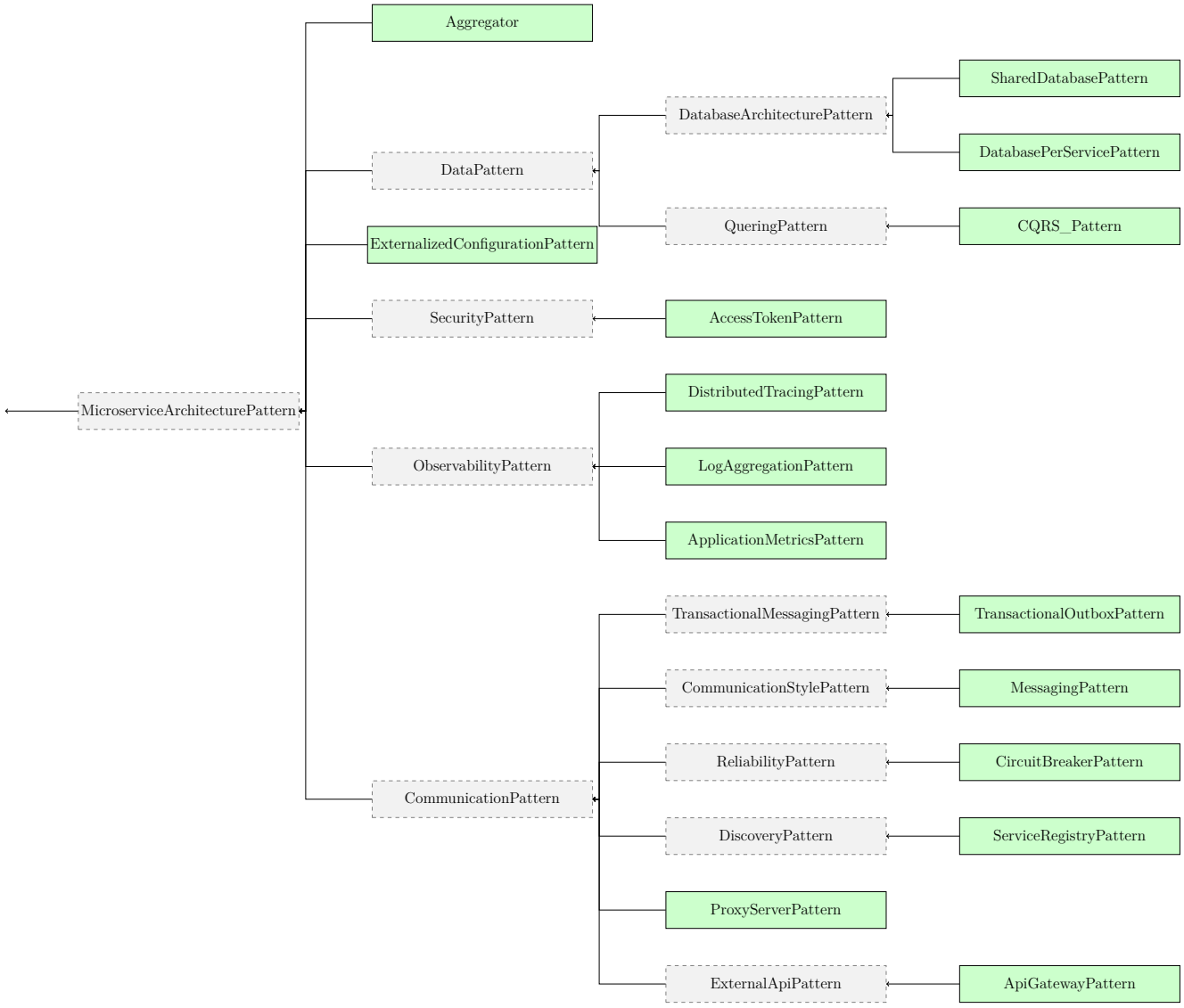


Figure 5.4: Classes derived from the `MicroserviceArchitecturePattern` class. *Concrete classes are green, abstract classes are grey.*

A large majority of microservice architectural patterns are centered around a particular service that carries the technical aspect of the pattern. The `msid` class property captures the identifier of this specific service. A distinct group of services are related to it as they pertain to the pattern and are captured in a set assigned as a value to the `sids` class property. The patterns usually encompass a number of properties that are either specific to the main service or the related services. To capture them, the `MicroserviceArchitecturePattern` introduces respectively the `main_service` and `service` class properties. Listing 5.4 illustrates a declaration of one of the most common patterns, service regitry, through the declaration of `ServiceRegistryPattern` concern.

```
1  ["Registry"]
2   = new ServiceRegistryPattern {
3      msid = "registry_service"
4      sids = Set("service_1", "service_2", "service_3")
5      main_service {
6          image = "image_location_of_the_registry"
7          expose = 1234
8      }
9      service {
10         image = "image_location_of_the_services"
11         environment{
12             ["REGISTRY_PORT"] = 1234
13         }
14     }
15     services {
16         ["service_1"] {
17             port = "80:80"
18         }
19     }
20   }
```

Listing 5.4: ServiceRegistryPattern example

As shown at *line 17* of Listing 5.4, the `services` property class is inherited from the `Concern` class and can be used to define properties for subsets of services contained in `sids`. Here we can think of `service_1` as the web frontend of the application whose public port is defined in the concern.

5.3.3 Service Centric vs Property Centric Concerns

The dichotomy between property and service centric concern is universal according to our observations. Whatever the aim of the architect, it falls in one or the other.

Service centric patterns fall by nature in the pattern category whether because they are an identified pattern or at least as a pattern specific to a particular application. Moreover, although it is not seemingly a necessity, all of the patterns that we integrated in the library, and that are to different extents well identified and shared by multiple applications, are service centric patterns.

Isolated property centric concerns have somewhat of an unsatisfying nature. Some of their instances make sense as they constitute a legitimate concern that profits from being isolated from the rest of the applications. It is for intance the case of the `ImageConcern` in Listing 5.1 where the design choice of naming the images through a common pattern is perfectly encapsulated and would be diluted if mixed with any other property. However the danger of designing, or rewriting, an application around property centric concerns is obvious when realizing that any application described using the common meta-model shown in Figure 4.1 can be factorized through property centric concerns, providing limited to zero architectural clarification.
An exemplar is the `expose` property, part of the `ServiceRegistryPattern`, in Listing 5.1 *line 14*. The decision to have it being part of the `ServiceRegistryPattern` is, although architecturally relevant, non-obvious as discussed in Section 4.2.2 but in any case undoubtedly better than to have it being a concern of its own as it would hold no architectural value.
This exemplifies a rule of thumb when writing a concern oriented description of a declarative system, that is that unless they have strong architectural value, property centric concerns should be as much as possible integrated into service centric concerns.

5.3.4 Respective Usefulness of Concerns Types

Both property centric concerns and service centric concerns have the beneficial effect of unifying scattered properties in architecturally meaningful structure. However, they represent two different but fundamental aspects of microservice applications. We present in Section 5.4 many examples of textual concerns implementations using the COMSA language. Before that we use the Open5gs application [36] to present examples of each concern type to show their immediate usefulness. The Compose file used to deploy the application is constituted of 223 unique properties spread without architectural structure in services descriptions who are in consequence tangled between different architectural patterns and policies.

Service centric concerns constitute the underlying topology of the application whose clear representation benefits immediately to understandability. The Open5gs application has an uncommon structure which is not reflected in the service names making it significantly harder to make sense of without relying on extensive knowledge of the application inner workings. Listing 5.5 shows that using the COMSA language, the topology of the application is made evident through the instantiation of service centric concerns whose Figure 5.5 shows in a graphical representation using the `topology` tool from the COMSA toolbox described in Section B.3.1 which produces a graph of the topology using only the COMSA description.

```
concerns {
  ["Around AMF CNF"]
    = new Aggregator {
        msid = "amf"
        sids = Set("ausf", "udm", "udr", "pcf", "bsf")
        // Properties...
      }

  ["Shared Mongo Database"]
    = new SharedDatabasePattern {
        msid = "mongo"
        sids = Set("scp", "pcf", "pcrf", "udr", "webui")
        // Properties...
      }

  ["Around NRF CNF"]
    = new SharedService {
        msid = "nrf"
        sids = Set("scp", "amf", "ausf", "bsf", "nssf", "pcf", "smf", "udm", "
            udr", "upf")
        // Properties...
      }

  ["Around SCP CNF"]
    = new SharedService {
        msid = "scp"
        sids = Set("amf", "ausf", "bsf", "nssf", "pcf", "smf", "udm", "udr", "
            upf")
        // Properties...
      }
  // Other concerns...
}
```

Listing 5.5: `ServiceCentricConcerns` in the Open5gs application encapsulating the topology.

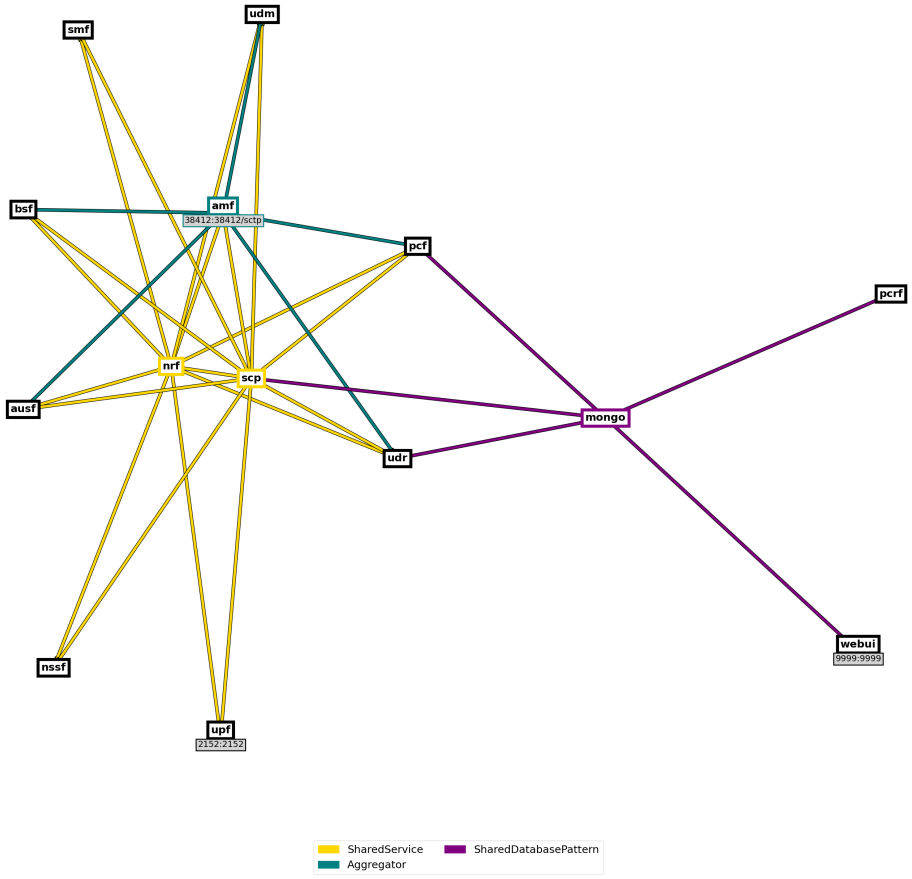


Figure 5.5: Representation of the topology of the Open5gs application.

Property centric concerns isolate the different policies enforced in the application. An interesting observation in the original Compose file is that the property `restart : on-failure` is present in all the microservices' descriptions but `mongodb` which is the only database in the application. The consequence of this possible oversight is a single point of failure making the application unusable without manual intervention if `mongodb` fails. Listing 5.6 shows the restart on failure policy implemented in COMSA through the declaration of a `RestartOnFailureConcern` object listing exhaustively the incorporated microservices.

```
1 ["Restart On Failure"]
2   = new RestartOnFailureConcern {
3       sids = Set("webui", "upf", "amf", "ausf", "scp", "bsf", "smf", "udr", "
            nssf", "udm", "pcf", "nrf", "pcrf")
4     }
```

Listing 5.6: Restart policy expressed through a `RestartOnFailureConcern` in the Open5gs application exhaustively listing incorporated microservices.

Even though the unification of the scattered property has been done through the use of the `Concern` object, the extension of the restart policy in the application is difficult to apprehend because of the large number of listed microservices (*line 3*). However, Figure 5.6 using the `hypergraph-policies` tool from the COMSA toolbox described in Section **??** which produces an hypergraph of the policies in the application using only the COMSA description, we can immediately identify the `mongodb` microservice not being integrated in the pattern.

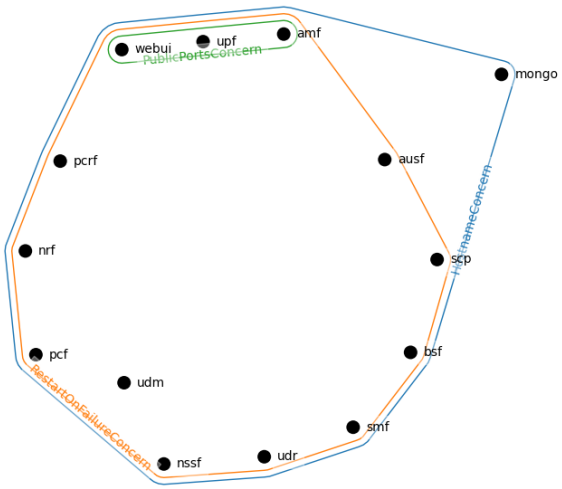


Figure 5.6: Representation of the policies in the Open5gs application.

Listing 5.7 shows an equally evident COMSA implementation through the use of the `ALL_SERVICES` keyword and Pkl `Set` method `difference`, highlighting that `mongodb` is not part of the policy. We can safely assume that had the application been written using a concern-based language that last part would not have been specified and the single point of failure avoided.

```
["Restart On Failure"]
  = new RestartOnFailureConcern {
      sids = ALL_SERVICES.difference("mongodb")
    }
```

Listing 5.7: Restart policy expressed through a `RestartOnFailureConcern` in the Open5gs application using the `ALL_SERVICES` keyword and `difference` Pkl method.

5.4 Overview of the library with examples from the dataset

5.4.1 Property Centric Concern examples

Property centric concerns are heterogeneous which is understandable since they are by definition the complete extension of microservice configuration for a given language. The following sections present architecturally concerns of different natures expressible through the instantiation of `PropertyCentricConcern` classes from Figure 5.3. They are accompanied by examples from the concern oriented dataset discussed in Section 6.1.
It is to be noted that a `PropertyCentricConcern` is not by essence tied to a particular concern except to the concern of focusing on that particular property. Because of that a `PropertyCentricConcern` derived class can be used to express a concern of one nature in an application description and of another in another description. It can be pertaining to both concerns in a third and to neither in a fourth.
A fabricated example could be made using the `ImageConcern` class that is often used to describe a naming pattern, can be part of a security concern, by gathering services having their images in a secure repository, or be part of both if the secure images have a common naming pattern or of neither while still being relevant for instance if multiple images simply have the same value.

Naming patterns

Naming patterns are considered good practice [1] [99] and are very frequent when describing microservices applications. Encapsulating naming patterns in concerns, especially using a function, allows to express the precise intent of the architect, removes a significant number of redundancies thus increasing readability and limitating error-proneness. Properties whose values are `String` and not limited to a number of language specific predefined options

are the most natural candidates. Some concerns commonly capturing naming patterns are `ImageConcern`, `EnvironmentConcern`, `ContainerNameConcern` or `HostnameConcern`.

```
["hostnameTemplate"]
= new HostnameConcern {
  sids = AllServices.difference(Set("nginx-web-server"))
  hostname = module.SID
}
```

Listing 5.8: HostnameConcern encapsulating a naming pattern.
source: Deathstarbench Media Microservices

Listing 5.8, extracted from the COMSA description of the Deathstarbench Media Microservices application [15] [33], shows the encapsulation of a naming pattern defining the `hostname` property for all but one service assigning the service identifier to the property. The `Concern` effectively merges 32 identically formated but scattered properties in the original file into one, and undoubtedly highlights the intent of the architect.

Runtime Policies

Some properties are meant to have a system wide functional effect, as such they are runtime policies. Policies can themselves be of different natures e.g., security policies, deployment policies, but specify those concerns in being universal to the services in the application, to the tolerance of a few justified exceptions. Properties having an effect on ressources or on runtime behavior of services are candidates for being policies concerns expressed through a `PropertyCentricConcern` derived class instance.

```
["Restart Always"]
  = new RestartAlwaysConcern {
      sids = ALL_SERVICES
    }
```

Listing 5.9: Runtime policy of restarting any service on faillure captured by a `RestartAlwaysConcern` instance.
source: Piggy Metrics

Listing 5.9, extracted from the COMSA description of the Piggy Metrics application [78], shows the encapsulation of the restart policy decided for the whole application through the usage of the macro `ALL_SERVICES`. The concern effectively merges identical scattered properties of 14 services in the original file into one, revealing explicitly the architect intent.

Security Concerns

Security concerns are less useful for factorizing or even expressing an architectural design that they are for providing awareness of critical points in the system. The factorizing effect and consequent increase in coherence can exist, particularly when the security concern is

also a policy, with `PropertyCentricConcern` derived classes such as `ReadOnlyConcern` or `SharedNetworkConcern`.

```
["PublicPorts"]
= new PublicPortsConcern {
  ports {
    ["ts-avatar-service"] {"17001:17001"}
    ["ts-admin-user-service"] {"16115:16115"}
    ["ts-admin-travel-service"] {"16114:16114"}
    ["ts-admin-route-service"] {"16113:16113"}
    ["ts-admin-order-service"] {"16112:16112"}
    ["ts-admin-basic-info-service"] {"18767:18767"}
    ["ts-consign-price-service"] {"16110:16110"}
    ["ts-consign-service"] {"16111:16111"}
    ["ts-food-service"] {"18856:18856"}
    ["ts-route-plan-service"] {"14578:14578"}
    ["ts-food-map-service"] {"18855:18855"}
    ["ts-voucher-service"] {"16101:16101"}
    ["ts-news-service"] {"12862:12862"}
    ["ts-ticket-office-service"] {"16108:16108"}
    ["ts-travel-plan-service"] {"14322:14322"}
    ["ts-seat-service"] {"18898:18898"}
    ["ts-assurance-service"] {"18888:18888"}
    ["ts-cancel-service"] {"18885:18885"}
    ["ts-rebook-service"] {"18886:18886"}
    ["ts-payment-service"] {"19001:19001"}
    ["ts-execute-service"] {"12386:12386"}
    ["ts-inside-payment-service"] {"18673:18673"}
    ["ts-security-service"] {"11188:11188"}
    ["ts-notification-service"] {"17853:17853"}
    ["ts-price-service"] {"16579:16579"}
    ["ts-ticketinfo-service"] {"15681:15681"}
    ["ts-basic-service"] {"15680:15680"}
    ["ts-preserve-other-service"] {"14569:14569"}
    ["ts-preserve-service"] {"14568:14568"}
    ["ts-travel2-service"] {"16346:16346"}
    ["ts-travel-service"] {"12346:12346"}
    ["ts-train-service"] {"14567:14567"}
    ["ts-station-service"] {"12345:12345"}
    ["ts-config-service"] {"15679:15679"}
    ["ts-order-other-service"] {"12032:12032"}
    ["ts-order-service"] {"12031:12031"}
    ["ts-contacts-service"] {"12347:12347"}
    ["ts-route-service"] {"11178:11178"}
    ["ts-verification-code-service"] {"15678:15678"}
    ["ts-user-service"] {"12342:12342"}
    ["ts-auth-service"] {"12340:12340"}
    [Dashboard] {"8080:8080"}
    [RabbitMQ] { "5672:5672" "15672:15672" }
  }
}
```

Listing 5.10: Public ports gathered in a single concern.
source: Train Ticket

Listing 5.10, extracted from the COMSA description of the Train Ticket application[32], does not aim at reducting the number of properties in the file but rather to gather all points of entry on the transport layer for the application. While it remains a challenge so sort through the different open ports, it is a significant improvement in readability

to have the 48 properties located consecutively in the same block, readable on a single screen, compared to the original file where those properties where scattered among the 272 properties of 68 services. The use of the `PublicPortsConcerns` to improve readability and focus on this critical matter can be fully exploited in combination with graphical tools such as the Hypergraph representation funished in the toolbox dand described in Section B.3.2, allowing to distinguish at a glance if a service has an open port or not.

Connection Concerns

The fundamental characterization of microservice applications that we adopted is that they are sets of units of executions communicating through the network to function as a whole. The latter however is observable through properties defined in a deployment file to allow for communication at runtime. Certain properties capture wide area of allowed communication that provides a high level understanding of the application architecture. Connection concerns can be considered security concern from another viewpoint as they characterize reachability and by contrast isolation.

```
1  ["Robotshop Network"]
2    = new SharedNetworkConcern {
3        sids = ALL_SERVICES
4        network_name = "robot-shop"
5      }
```

Listing 5.11: Network sharing among all microservices expressed through a `SharedNetworkConcern` instance.
source: Robot Shop

Listing 5.11, extracted from the COMSA description of the Robot Shop application [46], shows the application network being shared by all the services in the application implying that communication is possible between any two services. Would it be the case that some services were meant to be isolated, the inadequacy of the description would be immediately identifiable when the original description would have demanded to go through all 12 services properties to spot an intendedly absent one.

Procedural Concerns

While COMSA is a declarative language, as are most microservices ADLs, some microservices properties consist of procedural elements. For most properties of collection types, i.e., meant to contain multiple values, e.g., lists, sets, dictionaries, the order of elements has no impact on the functional value. Because of that any construction of services done by merging their partial definition in different concerns poses no difficulty. For instance, a service can have multiple environment variables defined in different `Concern` instances and any order for the ultimate real world service description produced through compilation to an execution language is equivalent to another - as long as there is no erroneous declaration

such as double declaration.
With the major factor of independence of properties, this secondary factor makes the *breakpoint problem* non existent for concern oriented languages as long as they compile to other declarative languages.
While properties with procedural values are still independent from other properties, their elements are not independent from one another.

```
["Healthcheck"]
  = new HealthcheckConcern {
      healthcheck = (sid) ->
        let (url =
             if(sid == "ratings")
               "http://localhost/_health"
             else if(sid == "web")
               "http://localhost:8080/"
             else // Regularity
               "http://localhost:8080/health"
            )
          new Healthcheck {
            test {
              "CMD"
              "curl"
              "-H"
              "X-INSTANA-SYNTHETIC: 1"
              "-f"
              url
            }
            interval = "10s"
            timeout = "10s"
            retries = 3
          }
      sids = Set("catalogue", "user", "cart", "shipping", "payment", "ratings
          ", "web")
    }
```

Listing 5.12: Heathcheck concern with procedural property.
source: Robot Shop

Listing 5.11, extracted from the COMSA description of the Robot Shop application[46], shows a `HealthcheckConcern` instance meant to represent a commong pattern for multiple services in the application. The functional effect is to periodically execute a command in the container encompassing the service to determine, on the basis of the command return value, if the service is "healthy" or not. A particular usage of that process is to inform the deployment tool if it can start other services that depend on the "heath" of another service or if it should wait for it to finish starting up.
The `healthcheck` attribute of the `HeathcheckConcern` instance, is assigned an `Healthckeck` instance, defined in the `compose.pkl` module, that has multiple class properties and among them the `test` class property *lines 13-20*. That attribute receives a `List` of `String` as value that is the command to be executed split around spaces making each continous series of characters an element. The order of elements being possibly determinant to the command execution, the list cannot be split among different `Concern`

objects without indicating their order for reconstruction, which brings back the breakpoint problem.
Instead of engaging in that direction, we chose to impose that properties whose values are collections with a non equivalent element order are to be considered as atomic and thus not splitted among different `Concern` instances. Aside from the practical aspect of that decision, it appears justified to set aside those properties as they conflict with the intended declarative nature of ADLs, which is illustrated by the fact that they constitute less that 2% of properties in our dataset.

5.4.2 Service Centric Concerns examples

Apart from the more general `ServiceCentricConcern` that is meant to capture service centric concerns that have not been identified and implemented in the library, service centric concerns are identified microservice architecture patterns that as shown in Figure 5.4 derive in the library from the `MicroserviceArchitecturePattern` class. Their structure is identical, allowing for generalization in the attributes names. The `msid` attribute designs the identifier of the pattern's central microservice and the `main_service` (alternatively `main_service_template`) attribute receives its properties that participate to the pattern. The same mechanics are used for the `sids` attribute, that receives a set of identifier designating the microservices in relation with the main service, and the `service` (alternatively `service_template`) receives the properties participating to the pattern which are assigned to each service.
The main service in identified patterns is always a technical service whose behavior is the core of the pattern that often implies certain relations between it and the client services and sometimes constrains. It allows tools from the COMSA toolbox, described in Chapter B, such as compilers and error checker to respectively inject properties in the executable code that are implicit in COMSA description and to check for inconsistencies between the application description and the use of the pattern.

`ServiceRegistryPattern`

A service registry[87] is a service to which services register in order to allow other services to discover them. The mechanism makes it not necessary for microservices to know the addresses of services they connect to. It thus allows for dynamic addressing resulting from failure or scaling. In addition to the discovery aspect, the service registry often effectuates health checks on registered services, then having the ability to inform other services of unavailabilities. The principal drawbacks are the increased development cost and the fact that it introduces a single point of failure in the infrastructure thought the former is largely compensated by the benefits resulting from implementation and the latter is still a better solution that the multiple single points of failure existing in a system where required addresses are hard coded in the program. Popular execution platforms such as

Docker or Kubernetes implement de facto system registry, allowing services to reach each other using only their identifiers, making service registry an implicit pattern.

```
["Consul"]
  = new ServiceRegistryPattern {
      msid = "consul"
      main_service {
        image = "consul:1.8.4"
        command = "consul agent -dev -client 0.0.0.0"
      }
      sids = BusinessServiceIds + Set("apache")
    }
```

Listing 5.13: `ServiceRegistryPattern` example.
source: Consul

Listing 5.13, extracted from the COMSA description of the Consul application [107] [106], shows a `ServiceRegistryPattern` instance representing the relation between the service registry whose identifier, i.e., `consul` is defined in the `msid` field and its associated properties in the `main_service` section. It is to be noted that its image and starting command are relevantly defined in this block. The client services are declared in the `sids` attribute as a `Set` of identifiers making use of `Pkl` set operations to add the `apache` service to the already defined `BusinessServiceIds` service identifiers set.

`ApiGatewayPattern`

An API gateway service[87] acts as the entry point in the application for all client connections. It thus provides all the endpoints reachable from outside the application. On reception of a client request, it has the responsibility to resolve it and answer. The resolution can be done by simple routing of the request to the appropriate back end service or more complex mechanism such as composition[76]. To the difference of service registry pattern, all services are not meant to be connected to the API gateway service but only those to which it transmits requests or interrogates to resolve clients requests.

```
["API Gateway"]
  = new ApiGatewayPattern {
      msid = "ftgo-api-gateway"
      main_service_template {
        environment {
          ["ORDER_DESTINATIONS_ORDERSERVICEURL"] = "http://ftgo-order-service
              :8080"
          ["ORDER_DESTINATIONS_ORDERHISTORYSERVICEURL"] = "http://ftgo-order-
              history-service:8080"
          ["CONSUMER_DESTINATIONS_CONSUMERSERVICEURL"] = "http://ftgo-consumer
              -service:8080"
        }
      }
      sids = Set("ftgo-order-service", "ftgo-order-history-service", "ftgo-
          consumer-service")
    }
```

Listing 5.14: `APIGatewayPattern` example.
source: Food To Go

Listing 5.14, extracted from the COMSA description of the Food To Go application [86] [87], shows an `ApiGatewayPattern` using the `msid` field to designate `ftgo-api-gateway` as the gateway service, and `sids` field to designate client services using a set of identifiers. The field `main_service_template` is used to define environment variables for the API gateway service.

`DistributedTracingPattern`

A tracing service[87] is a service that gathers tracks of requests through the application. It is an observation pattern that aims at understanding for specific requests which microservices are sollicited what is their actions. This pattern is particularly useful for large applications where the independant nature of microservices, in their functionning as well as in their development, makes the static analysis factually impossible. Though useful, this pattern is invasive and requires instrumentation of every microservice in order for them to transmit to the tracker service the relevant informations, that also have to be defined by the developper. At minimum each request has to be marked with an identifier that is transmitted in all messages, those being numbered to keep track of order.

```
["Distributed Tracing Pattern"]
  = new DistributedTracingPattern {
      msid = "tracing-server"
      main_service_template {
        image = "openzipkin/zipkin"
        environment = new Listing {
          "JAVA_OPTS=-XX:+UnlockExperimentalVMOptions -Djava.security.egd=
             file:/dev/./urandom" }
      }
      sids = ServiceSids + Set("api-gateway")
      service_template {
        environment = new Listing {
          "SPRING_ZIPKIN_BASEURL=http://" + msid + ":9411"
        }
      }
    }
```

Listing 5.15: `DistributedTracingPattern` example.
source: Pet Clinic

Listing 5.15, extracted from the COMSA description of the Pet Clinic application[91], shows a `DistributedTracingPattern` using the `msid` field to designate `tracing-server` as the tracer service, and the `sids` field to designate traced services using a set of identifiers. The field `main_service_template` is used to define the image of the tracer service as well as an environment variable indicating options for the Java Virtual Machine that executes it. The field `service_template` is used to define an environment variable indicating the address where to send the data about treated queries. As the application also uses a service registry pattern, services can communicate using services identifiers and so the tracer address is a function of the main service identifier (`msid`).

`MessagingPattern`

A message broker is a microservice that handles messages in a centralized manner to allow for asynchronous communication between microservices. Without implementation of a messaging pattern communication between microservices is generally done in a *one-to-one* synchronous manner between microservices where a requester microservice sends a message to another microservice and waits for response. A messaging paradigm is asynchronous, it allows also for a request/response model but permits also simple notifications without responses as well as *one-to-many* transmissions either with notifications (publish/subscribe) or asynchronous responses.

```
["Messaging Pattern"]
  = new MessagingPattern {
      msid = "rabbit"
      main_service_template {
        image = "rabbitmq:3-management"
        hostname = "rabbit"
        environment {
          ["RABBITMQ_ERLANG_COOKIE"] = "SWQOKODSQALRPCLNMEQG"
          ["RABBITMQ_DEFAULT_USER"] = "guest"
          ["RABBITMQ_DEFAULT_PASS"] = "guest"
          ["RABBITMQ_DEFAULT_VHOST"] = "/"
        }
        labels {
          ["NAME"] = "rabbitmq"
        }
      }
      sids = Set("uc", "intention", "position", "order", "api-gateway", "
         zipkin")
      service_template {
        environment = new Listing {
          "RABBIT=rabbit"
        }
      }
    }
```

Listing 5.16: `MessagingPattern` example.
source: QBike

Listing 5.16, extracted from the COMSA description of the Q Bike application[7], shows a `MessagingPattern` using the `msid` field to designate `rabbit` as the message broker service, and the `sids` field to designate its client services using a set of identifiers. The field `main_service_template` is used to define environment variables relative to manage main service, i.e., the message broker. The field `service_template` is used to set an environment variable for all client services that indicates the identifier of the message broker which is sufficient since the application uses a service registry pattern.

`CircuitBreakerPattern`

The resilience of microservice applications comes from the independent and often stateless nature of business microservices that permits redundancy and inconsequential restarting of services. Although entire unavailability of the entire system is less probable than it is with monolithic applications, partial failures are still a non negligible hazard. For applications using synchronous communications, aside from the immediate consequences in terms of functionalities of the application, the risk of a cascading effect on healthy services comes from them blocking in wait of a response. The circuit breaker pattern mitigates that effect by enforcing a timeout on connection directed toward unresponsive microservices during which it denies all requests that would imply a request to them. That mechanism, is a simple mean to contain failures and keep the functionalities not impacted by the local failures available.

```
["Circuit Breaker"]
  = new CircuitBreakerPattern {
      msid = "turbine-stream-service"
    }
```

Listing 5.17: `CircuitBreaker` example.
source: Piggy Metrics

Listing 5.17, extracted from the COMSA description of the Piggy Metrics application [78], shows a `CircuitBreakerPattern` instance using the `msid` field to designate `turbine-steam-service` as implementing the CircuitBreakerPattern. As the `sids` field is not set, it implies that the pattern is applied to every service it connects to.

`SharedDatabasePattern`

The ideal structure to preserve independence and statelessness between microservices is to have separate dedicated databases for each microservice. However it comes at least with greater cost in architecting the application and happens to be outwright impossible in some cases where data managed by different services is too tightly coupled to not have shared databases. It is then important to identify which microservices are linked to a particular database.

```
["Shared Mongo Database"]
  = new SharedDatabasePattern {
      msid = "mongo"
      main_service_template {
        image = "mongo"
        environment {
          ["MONGO_INITDB_DATABASE"] = DATABASE_NAME
        }
        expose {
          "27017/udp"
          "27017/tcp"
        }
        healthcheck {
          test {
            "CMD"
            "mongosh"
            "--eval"
            "db.adminCommand('ping')"
          }
          interval = "5s"
          timeout = "5s"
          retries = 3
        }
      }
      sids = Set("scp", "pcf", "pcrf", "udr", "webui")
      service_template {
        environment {
          ["DB_URI"] = "mongodb://\(msid)/\(DATABASE_NAME)"
        }
      }
    }
```

Listing 5.18: `SharedDatabasePattern` example.
source: Open 5gs

Listing 5.18, extracted from the COMSA description of the Open 5gs application [36], shows a `SharedDatabasePattern` instance using the `msid` field to designate `mongo` as the shared database. The field `main_service_template` is used to document the database exposed port and set an healthcheck to inform other services of the database status. The `sids` fields determines the services among which the database is shared using a set of identifiers. The field `service_template` is used to set an environment variable for all services using the database providing the access point to it.

`DatabasePerServicePattern`

If a microservice can isolate its managed data from other microservices, it is advisable to dedicate a database to it as it makes it completly independent from other services from development and execution standpoints. However if the patterm is consistently applied to an application composed of many services, managing the service-database couples can be a source of confusion. The `DatabasePerServicePattern` makes it simple to identify couples and configure them as a unit.

```
["userSQLDB"]
= new DatabasePerServicePattern {
  sid = "user-service"
  msid = "user-mongodb"
}
```

Listing 5.19: `DatabasePerServicePattern` example.
source: Deathstarbench Social Network

Listing 5.19, extracted from the COMSA description of the Deathstarbench Social Network application [16] [33], shows a `DatabasePerServicePattern` instance using the `msid` field to designate `user-mongodb` as the dedicated database to the service `user-service` designated in the `sid` field. While using the more general `sids = Set("user-service")` is still possible, as, by design, only one service can be assignated to the database, a more natural singular field has been defined for this particular `Concern`. The `DatabasePerServicePattern` instance highlights the connection between the two services and and could be used to configure both the business service and database using the `main_service`, `service` or `services` fields.

5.4.3 Multiconcerns

Certain applications are characterized by the repetition of a specific pattern. In the same manner that concerns capture the commonality in property settings through the `Concern` class in the presented implementation of the COMSA language, we thought of proposing an higher order concern to capture repeating concerns in an application.

```
  ["homeTimelineNoSQLDB"]
  = new DatabasePerServicePattern {
    sid = "home-timeline-service"
    msid = "home-timeline-redis"
  }
  ["socialGraphSQLDB"]
  = new DatabasePerServicePattern {
    sid = "social-graph-service"
    msid = "social-graph-mongodb"
  }
  ["socialGraphNoSQLDB"]
  = new DatabasePerServicePattern {
    sid = "social-graph-service"
    msid = "social-graph-redis"
  }
  ["postStorageSQLDB"]
  = new DatabasePerServicePattern {
    sid = "post-storage-service"
    msid = "post-storage-mongodb"
  }
  ["userTimelineSQLDB"]
  = new DatabasePerServicePattern {
    sid = "user-timeline-service"
    msid = "user-timeline-mongodb"
  }
  ["userTimelineNoSQLDB"]
  = new DatabasePerServicePattern {
    sid = "user-timeline-service"
    msid = "user-timeline-redis"
  }
  ["urlShortenSQLDB"]
  = new DatabasePerServicePattern {
    sid = "url-shorten-service"
    msid = "url-shorten-mongodb"
  }
  ["userSQLDB"]
  = new DatabasePerServicePattern {
    sid = "user-service"
    msid = "user-mongodb"
  }
  ["mediaSQLDB"]
  = new DatabasePerServicePattern {
    sid = "media-service"
    msid = "media-mongodb"
  }
  ["mediaMemcached"]
  = new DatabasePerServicePattern {
    sid = "media-service"
    msid = "media-memcached"
  }
  ["userMemcached"]
  = new DatabasePerServicePattern {
    sid = "user-service"
    msid = "user-memcached"
  }
  ["urlShortenMemcached"]
  = new DatabasePerServicePattern {
    sid = "url-shorten-service"
    msid = "url-shorten-memcached"
  }
  ["postStorageMemcached"]
  = new DatabasePerServicePattern {
    sid = "post-storage-service"
    msid = "post-storage-memcached"
  }
```

Listing 5.20: `DatabasePerServicePattern` repeated multiple times in an application.
source: DeathstarBench Social Network

Listing 5.19 shows an instance of a `DatabasePerServicePattern` that only one of many present the COMSA description of the Deathstarbench Social Network application as shown in Listing 5.20. While it is a considerable improvement from the original Docker Compose file describing the application that suffered from the scattering of properties and absence of architectural informations, the repetitiveness of this section can be reduced.

```
["DatabasePerServicePatterns"]
= new MultiDatabasePerServicePattern{
  attributes = {"msid" "sid"}
  values {
    "post-storage-memcached" "post-storage-service"
    "url-shorten-memcached"  "url-shorten-service"
    "user-memcached"         "user-service"
    "media-memcached"        "media-service"
    "media-mongodb"          "media-service"
    "user-mongodb"           "user-service"
    "url-shorten-mongodb"    "url-shorten-service"
    "user-timeline-redis"    "user-timeline-service"
    "user-timeline-mongodb"  "user-timeline-service"
    "post-storage-mongodb"   "post-storage-service"
    "social-graph-redis"     "social-graph-service"
    "social-graph-mongodb"   "social-graph-service"
    "home-timeline-redis"    "home-timeline-service"
  }
}
```

Listing 5.21: `MultiDatabasePerServicePattern` example.
source: Deathstarbench Social Network

Listing 5.21 shows all the `DatabasePerServicePattern` instances of Listing 5.20 gathered into a single `MultiConcern` instance. Here the nature of the `Concern` of which multiple instances are defined in a single declaration is implicit as the object instantiated is a specific class, `MultiDatabasePerServicePattern`, derived from the `MultiConcern` class. The `attributes` field could also be omitted as it is by default defined this way in the derived class but it's explicit indication of the order of the services provides better readability.

`MultiConcern` can be used for any type of concern in a simple manner, even without a derived class. Listing 5.22 instantiation is equivalent to the one in Listing 5.21 but uses only the `Multiconcern` class, showing how to use it for any concern when useful to the description.

```
  ["DatabasePerServicePatterns"]
  = new MultiConcern{
    concern = module.DatabasePerServicePattern
    attributes = {"msid" "sid"}
    values {
      "post-storage-memcached" "post-storage-service"
      "url-shorten-memcached"  "url-shorten-service"
      "user-memcached"         "user-service"
      "media-memcached"        "media-service"
      "media-mongodb"          "media-service"
      "user-mongodb"           "user-service"
      "url-shorten-mongodb"    "url-shorten-service"
      "user-timeline-redis"    "user-timeline-service"
      "user-timeline-mongodb"  "user-timeline-service"
      "post-storage-mongodb"   "post-storage-service"
      "social-graph-redis"     "social-graph-service"
      "social-graph-mongodb"   "social-graph-service"
      "home-timeline-redis"    "home-timeline-service"
    }
  }
```

Listing 5.22: `MultiConcern` example defining multiple `DatabasePerServicePattern` in a single instantiation.
source: Deathstarbench Social Network

5.5 Implementation

This sections builds on Section 4.3.4 to delve into the Pkl implementation of the COMSA language.

5.5.1 `PropertyCentricConcern`

The `PropertycentricConcern` class definition simply adds the `sids` property class to the `Concern` class as it at the heart of the concern to designate a group of service sharing sets of coherent properties. Listing 5.23 shows that simple definition.

```
abstract class PropertyCentricConcern extends Concern {
  sids: Set<Id>?
}
```

Listing 5.23: Pkl implementation of the `PropertyCentricConcern` class

The actual mechanism of property assignation is done in the `SinglePropertyCentricConcern` class that derives from the `PropertyCentricConcern` class. The reason for that intermediate class is that it is dedicated to defining a single property for multiple services. Property centric concerns are not by nature limited to one property although we found it to be the most relevant usage as multiple properties defined for a set of microservices tend to be more relevant in functional pattern.

```
abstract class SinglePropertyCentricConcern extends PropertyCentricConcern {
  hidden property_name: String
  hidden property_value: Any
  services {
    when (sids == ALL_SERVICES) {
      [sids] = Map(property_name, property_value).toTyped(module.Service)
    }
    when (sids != ALL_SERVICES) {
      for (sid in sids ?? Set()) {
        [sid] =
        if (property_value is Function)
          Map(property_name, property_value.apply(sid)).toTyped(module.Service)
        else
          Map(property_name, property_value).toTyped(module.Service)
      }
    }
  }
}
```

Listing 5.24: Pkl implementation of the `SinglePropertyCentricConcern` class

Listing 5.24 shows the definition of the class used in all the examples presented in Section 5.4.1. To generalise the mechanics of property assignation to the individual services in the mother class, the specific targeted property is given in the `"property_name"` field. The value affected to the property is located in the `property_value` field. Its type is not restricted not to presume on the possible values of thereby set properties. The attribution logic makes use of the `services` inherited from the `Concern` class. It distinguishes between the property value being a function and any other type. In the latter case it simply assigns the value by creating an entry in the `services` mapping using the service identifier as key and as value a Compose Pkl `Service` with only the specific value set. In the former case, it applies the function on the service identifier in order to produce the property value. At time of writing, the function can only be instantiated with the service identifier as it happens to cover all the cases of the dataset. Both of those cases make use of the `toTyped` Pkl function that creates an instance of a class implementing the method from a map whose keys are attributes names and keys attributes values.

```
open class ImageConcern extends SinglePropertyCentricConcern {
  hidden property_name: "image"
  hidden property_value = this.image
  hidden image: (String|Function)?
}
```

Listing 5.25: Pkl implementation of the `ImageConcern` class

Listing 5.25 shows the definition of the `ImageConcern` concrete class used to instantiate property centric concerns setting the image property for sets of concerns. A new attribute `image` is declared to make the use of the concern intuitive and its possible value is restricted to either string or a function. The property name and value are respectively copied in the `property_name` and `property_value` for instantiating the service in the mother class.

5.5.2 ServiceCentricConcern

The service centric concerns shown in Section 5.4.2 are all of the `MicroserviceArchitecturePattern` class, meaning that they are all patterns identified as being commonly used in microservice applications.

```
abstract class MicroserviceArchitecturePattern extends Concern {
  sids: Set<Id>
  msid: Id
  behaviors: Listing<Behavior>
  service_template: Service = apply_behaviors(this, behaviors.toList())[sids]
  main_service_template: Service = apply_behaviors(this, behaviors.toList())[
      msid]
  services {
    when (!sids.isEmpty) {
      [sids] = service_template
    }
    when (!(msid is Null)){
      [msid] = main_service_template
    }
  }
}
```

Listing 5.26: Pkl implementation of the `MicroserviceArchitecturePattern` class (simplified)

Listing 5.26 shows the implementation of the `MicroserviceArchitecturePattern` class which is the common mother class for all the identified patterns presented in Section 5.4.2. The class has two analogous mechanisms, one for the main service and another for the other services connected to it. The identifier is assigned in the `msid`, respectively `sids`, class property and the properties are contained in the `main_service_template`, respectively `service_template` attribute. A list of behaviors can be added to the concern that are applied on top of the properties defined in the concrete class definition of a pattern and on those defined at instantiation in an application description. A mechanism to permit alternative names for the main or other services exists but has been omited in Listing 5.26 for clarity. The apply behavior function applies each `Behavior` object to the `Service` object that represents the main service, respectively related services, properties pertaining to the concern. The action of the apply function of `Behavior` is to add properties to services considered as being implied by the use of the concern. For instance, it is implied by the use of the `ServiceRegistryPattern` that for deployment, registered services depend on the service registry thus the `ServiceRegistryPattern` class definition has the corresponding behavior by default as shown is Listing 5.27.

```
class ServiceRegistryPattern extends DiscoveryPattern {
  msid: Id
  behaviors {sids_depend_on_msid}
}
```

Listing 5.27: `ServiceRegistryPattern` class definition.

The default behavior of the pattern can however be modified by explicitly assigning an alternative value to the `behavior` attribute. For instance, to not have any implicit dependency, adding the line `behavior = new Listing{}` in the instance declaration will make it so that no behavior is applied. Alternative behaviors can be declared instead. The list of available behaviors is given in Listing 5.28.

```
sids_depends_on_msid            // alternatively sids_depend_on_msid
sids_depends_on_started_msid    // alternatively sids_depend_on_started_msid
sids_depends_on_healthy_msid    // alternatively sids_depend_on_healthy_msid

msid_depends_on_sids            // alternatively msid_depend_on_sids
msid_depends_on_started_sids    // alternatively msid_depend_on_started_sids
msid_depends_on_healthy_sids    // alternatively msid_depend_on_healthy_sids

sids_connect_to_msid            // indicates relation, no functional effect,
    used for graphical representation
```

Listing 5.28: List of available behaviors for `MicroserviceArchitecturePattern` classes.

5.5.3 Multiconcern

The `MultiConcern` class gathers separate collections of data used to create a series of of instances of a particular `Concern` class.

```
open class MultiConcern{
  hidden concern: Class
  attributes: Listing<String>
  values: Listing<Listing | Dynamic | List<Id|Set<Id>> | Id | Set<Id> > | List<
      List<Id|Set<Id>>> | Set<List<Id|Set<Id>>> = new Listing{}
  main_service: Service = new Service{}
  service: Service = new Service{}

  function compute_concerns(): Mapping<String, Concern> =
  // ...
}
```

Listing 5.29: Pkl implementation of the `MultiConcerns` class (simplified)

The central mechanism of the class is located in the `compute_concerns` that resolves `MultiConcern` instances into a `Mapping` of `String`-`Concern` pairs, making them seamlessly integrable into otherwise declared concerns. The value attribute accepts any collection type for declaring the identifiers of participating services. The additional properties for the main and related services are assumed to be identical for all concerns and can be declared respectively in the `main_service` and `service` in the same manner as in single `Concern` declarations.

5.6 Conclusion

We have established a library of reusable concerns in the COMSA language, allowing explicit architectural description by design. Doing so we highlighted a fundamental distinction between property centric concerns and service centric concerns. The former covers concerns of different natures such as application policies, security concerns or naming patterns but have in common to relate to the application as a whole. The later characterizes architectural patterns and introduces a characterization of technical (or infrastructural) microservices as being at the center of a pattern. Because those patterns have an identified behavior they allow to make some of the properties implicit, symplifying the description.

An ADL is only as practically useful as its associated toolbox allows it to be and so we dedicate Chapter B to the presentation of the tools we developed to support the COMSA language. We rhen go on with the evaluation of the COMSA language in Chapter 6, to show that having an explicit architectural description by design provides gains in limiting greatly redundancy thus reducing error proneness by making the description more sound through unification of duplicated properties produced by the commonly shared metamodel.

Chapter 6

Evaluation

This chapter presents the quantitative evaluation of the COMSA language and more generally of the Concern concept through the rewriting of microservice applications initially written in the Compose language and the analysis of subsequent data.

6.1 Dataset presentation

The established dataset is a selection of 21 out of 61 applications from Davide Taibi's Curated Dataset of Microservice-Based Systems [81] [94]. We added to this selection the three finished applications from the DeathStarBench benchmark suite [17] [33].
Table 6.1 shows for each application the number of microservices as well its popularity through the number of stars and forks on Github. It shows the wide range of application complexity going from having only 5 microservices for the smallest (consul) to 68 for the largest (train-ticket) and from begin a personal project (ex: blog-microservices) to having 55 contributors (sockshop). In both the aspects of microservice number and contributor number, the dataset is evenly distributed with average values of respectively 15.4 and 14.4 and corresponding standard deviations of 13.8 and 14.4. Those attributes of the dataset coupled with the median of microservice number being 11 and the median of contributor number being 13 indicates that the core of the dataset is composed of middle sized projects that are in that regard representative of realworld applications. The total number of microservices being 324 and contributors to these applications being 303 indicate that the following analyses are applied on a wide range of types of microservices and preferences in terms of possible development approaches.
Although microservice applications quality cannot be assessed through their popularity, the median values for stars and forks being respectively 658 and 294, indicate that the dataset is not composed of niche or irrelevant projects and are considered worthy of attention by the microservice community.

Application	Microservice Number	Contributors	Stars	Forks
blog-microservices [27]	11	1	407	303
blogpost [6]	9	1	14	14
consul [106]	5	4	106	58
CQRS [104]	7	1	658	294
hotelreservation [14]	19	34	819	449
mediamicroservices [15]	33	18	819	449
socialnetwork [16]	27	28	819	449
ftgo-application [86]	16	3	3.6k	1.4k
graphprocessing [4]	9	3	17	14
LakesideMutual [69]	9	5	178	110
microcompany [26]	11	4	351	139
open5gs [36]	14	17	64	27
PetClinic [91]	10	38	1.8k	2.4k
PiggyMetrics [78]	14	13	13.5k	6.2k
Pitstop [103]	13	18	1.1k	481
qbike [7]	10	1	506	176
robot-shop [46]	12	16	921	4.8k
sockshop [18]	15	55	3.7k	2.9k
TapAndEat [31]	5	1	8	8
teastore [20]	7	19	132	156
train-ticket [32]	68	23	769	265
Total	324	303	30k	21k
Min	5	1	8	8
Max	68	55	13.5k	6.2k
Average	15,4	14.4	1.4k	1k
Median	11	13	658	294
Standard Deviation	13.8	14.4	2.8k	1.6k

Table 6.1: Dataset Composition and Base Data
stars and forks updated on 17-06-2025

6.2 Unique Properties

Section 4 and Section 5 present the qualitative improvement that the COMSA language provides in comparison to the common model of other microservice application's ADLs. These qualitative attributes we pretend to ameliorate are by nature hard to capture in a quantitative manner. One approach is to look at the number of lines of code generated from the new language and compare that value to the number of lines to be added for the application to function. Since one of the imperatives we imposed on ourselves in

the creation of the COMSA language was that it would be immediatly executable after compilation, so that there wouldn't be a discrepancy between source code and generated code, and that we achieved that goal, that approach is not applicable. Another approach is to ask a group of people to use the language and rate it in different aspects. It brings multiple difficulties relative to the sample of people chosen to perform the test for it to be representative of a larger scale. The possibility to gather a sufficiently large group of willing participants is in itself a challenge, then dealing with the variability in competence also poses a complex difficulty. Ultimately even assuming the previously mentioned difficulties have been overcome, the analysis resulting from such a survey objectify the qualitative attributes of a language but do not give a quantitative measure for it. One simple approach could have been to merely compare the number of lines between the original Compose files and the COMSA files. However that comparison would rely for the most part on the specific writing style of the author of the original file as well as on ours. Moreover the shortness of the file does not indicate a gain in the aspects we're aiming for which are understandability, error resilience, maintainability and coherence, as the possibility of transforming description files into one liners obviously shows. Counting characters does not offer a better alternative since having every user decided variable be of minimal length would negatively affect understandability, maintainability, error resilience and have no effect on coherence.

We chose to compare the number of unique properties between original Compose files and COMSA files as an objective measurement of qualitative improvement. We define an unique property in a microservice application description as the independent affectation of an attribute-value couple for one or multiple microservices. The affectation is independent in the sense that modifying the value in one place has no effect on any other affectation as exemplified on the Teastore application in Section 4.1.4. For instance Listing 6.1 shows two identical properties that are not one unique property since modifying the value of one does not affect the other.

```
service_1:
    image: my_image
service_2:
    image: my_image
```

Listing 6.1: Compose identical but independent properties.

Using variable instead of hard coded values does not create more coherence but only provides a mean to change multiple independent properties in a single place. While that mechanism increases maintainability, it also tends to harm understandability as it becomes necessary to look in a different place to consult the assigned value. It also does not differentiates services sharing the common value from those who do not and thus necessitate to exhaustively check each microservice definition in the file, thus limiting improvement in error resiliency.

On the other hand, the COMSA language allows for complete coherence effectively associating multiple microservices to a single property as shown in Listing 6.2. Modifying the `Image` attribute value affects all of the microservices that are explicitly listed in the `sids` attribute. Thus there is no need to look anywhere else in the file to ensure the range of the modification.

```
["Image Concern"]
= new ImageConcern{
    sids= Set{"service_1", "service_2"}
    image= "my_image"
}
```

Listing 6.2: COMSA unique property shared among multiple services.

The reduction of unique properties is a direct measurement of the gain in coherence but also indicates gains in harder to quantify values. Maintainability gains can be immediately deduced as there is less places to modify, from the reduction of entities comes the reduction in error-proneness as there is less places to check for and to make mistakes at. In the same manner, even without considering the arguments presented in Section 4 and Section 5 and the conceptual relevance of Concerns for microservice architecture, eliminating superfluous properties asks for less information to be considered thus increasing the ease of understanding.

Table 6.2 shows values for each applications for properties in original Compose files, similarities between properties in those files, unique properties in original Compose files and properties in COMSA descriptions of the same applications.

"Compose Properties Number" is the count of properties in Compose files considered in the sense defined earlier. Intuitively a property is any occurrence of an attribute-value assigned in a microservice description. Two properties are considered similar if they are identical modulo the service identifier thus each property that can be associated with another through that definition is added to the count of "Property similarities". "Unique properties" are then the count of each distinct property in the sense that for each group of similar properties count as one property. "COMSA Properties" is the count of properties in COMSA files. For "Compose Property Similarities", "Unique Properties" and "COMSA Properties", values are expressed absolutely and as a percentage of the corresponding value in the "Compose Property Number" column. Conversely the proportion of properties in the original Compose files relatively to "Unique Properties" and "COMSA Properties" is expressed in the gain column of those sections and give a metric for respectively the theoretical possible gain and actual gain through writing of the application description in the COMSA language. The "Diff" column indicates the difference between the number of unique properties and the number of properties in COMSA files.

Application	Compose Property Number	Compose Property Similarities		Unique Properties			COMSA Properties			Diff
		#	%	#	%	gain	#	%	gain	
blog-microservices	61	42	68,85	24	39,34	2,54	24	39,34	2,54	0
blogpost	63	40	63,49	32	50,79	1,97	22	34,92	2,86	-10
consul	14	7	50,00	9	64,29	1,56	7	50,00	2,00	-2
CQRS	38	22	57,89	23	60,53	1,65	19	50,00	2,00	-4
hotelreservation	157	119	75,80	56	35,67	2,80	47	29,94	3,34	-9
mediamicroservices	119	92	77,31	33	27,73	3,61	23	19,33	5,17	-10
socialnetwork	134	101	75,37	41	30,60	3,27	24	17,91	5,58	-17
ftgo-application	192	92	47,92	120	62,50	1,60	93	48,44	2,06	-27
graphprocessing	50	17	34,00	39	78,00	1,28	35	70,00	1,43	-4
LakesideMutual	46	29	63,04	25	54,35	1,84	22	47,83	2,09	-3
microcompany	47	28	59,57	23	48,94	2,04	23	48,94	2,04	0
open5gs	183	147	80,33	56	30,60	3,27	56	30,60	3,27	0
PetClinic	60	42	70,00	26	43,33	2,31	24	40,00	2,50	-2
PiggyMetrics	79	72	91,14	17	21,52	4,65	17	21,52	4,65	0
Pitstop	78	46	58,97	40	51,28	1,95	34	43,59	2,29	-6
qbike	80	50	62,50	40	50,00	2,00	35	43,75	2,29	-5
robot-shop	102	88	86,27	27	26,47	3,78	19	18,63	5,37	-8
sockshop	133	105	78,95	43	32,33	3,09	43	32,33	3,09	0
TapAndEat	10	4	40,00	7	70,00	1,43	6	60,00	1,67	-1
teastore	28	23	82,14	9	32,14	3,11	9	32,14	3,11	0
train-ticket	272	222	81,62	57	20,96	4,77	56	20,59	4,86	-1
Total	1946	1388	71,33	747	38,39	2,61	638	32,79	3,05	-109
Min	10	4	34,00	7	20,96	1,28	6	17,91	1,43	-27
Max	272	222	91,14	120	78,00	4,77	93	70,00	5,58	0
Average	92,7	66,1	66,91	35,6	44,35	2,60	30,4	38,09	3,06	-5,2
Median	78	46	68,85	32	43,33	2,31	24	39,34	2,54	-3
Standard Deviation	66,4	53,4	15,34	24,4	16,52	1,02	20,2	14,30	1,30	6,7

Table 6.2: Comparison of unique properties between original Compose files and COMSA files.

Table 6.2 confirms the size diversity of microservice applications in the dataset through number of properties as Table 6.1 did through numbers of microservices and collaborators. The dataset is well distributed in that regard as shown by the average number of properties being 92.7 and the standard deviation being 66.4 with the range of properties going from 10 to 272. The median number of properties being 78 gives an

idea of the number of properties in a middle sized microservice application written in Compose.

The "Compose Property Similarities" column indicates an high similarity even for the base value of 34% (application: graphprocessing) and shows extreme redundancy in the most severe case with 91.14% of properties being similar to another (application: PiggyMetrics). We can note that the application showing with the highest value is also by large the most popular of the dataset with 13k stars and 6.2k forks, thus highly reviewed, which hints toward a situation that is not a developper mishaps but a model ingrained problem. The median and the average are close to each other respectively 66.91% and 68.85% with a moderate standard deviation of 15.34 points indicating that on average only a third of the properties do not suffer from unecessary redundancy.

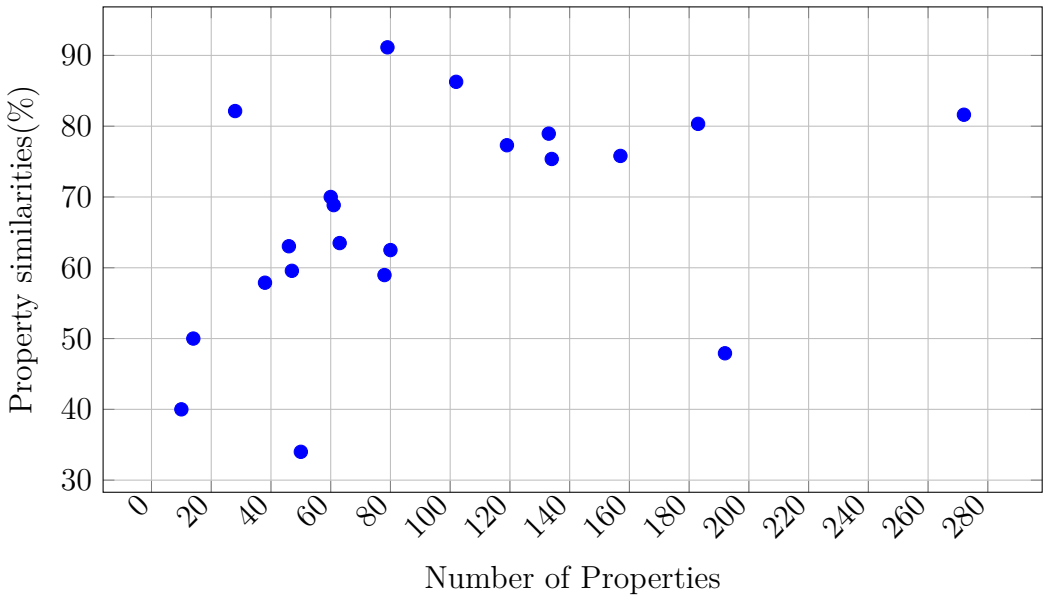


Figure 6.1: Relationship between number of properties and property similarities (%).

We observe a correlation between property number and property similarity occurences proportion in Table 6.2 that is shown in Figure 6.1 which puts in relation number of properties as the absissa and as the ordinate percentage of properties similar to at least another in the sense defined earlier.

The "Unique Property" column indicates considerable variability between Compose application descriptions with a minimum of only 20.96% of the properties being unique and a maximum of 78% unique properties with close average and median value of respectively 44.35% and 43,33% and a considerable standard deviation of 16.52 points.

On a global scale comparing the total properties present in the dataset and the total unique properties, we find that a gain of 2.61 is theoritically achievable, meaning that the dataset is expressed using 2.61 times the strictly necessary properties to produce an

equivalent global set of properties. That figure is conforted by the close values of average and median gain, respectively of 2.6 and 2.3 and a considerable but reasonable standard deviation of 1.02. From those values we conclude that it is reasonable to expect that Compose files can be expressed using only half the properties they're using by adopting a model allowing for assignation of properties to multiple services at once.

Figure 6.2: Relationship between number of properties and unique properties (%).

We observe an inverse correlation between property number and unique properties proportion in Table 6.2 that is shown in Figure 6.2 which puts in relation number of properties as the abcissa and as the ordinate percentage of unique properties in the sense defined earlier.

The "COMSA Properties" column presents the values relative to the COMSA written description of the original application describded in Compose. Each of the COMSA description is at least equivalent to the Compose description, meaning that if a unique property is expressed in the Compose file it is expressed explicitely or implicitely in the COMSA file and that it is explicitely observable in a generated Compose description through the use of the `comsa2compose` or `comsa2compose-yaml` compilers presented in Section B. The properties are counted in absolute values as well as proportionaly, as a percentage, to the original Compose property number for the same application. The "gain" column represents the inverse relation and gives a metric to estimate the gain accomplished by using the COMSA language instead of the Compose language.

On a global scale, expressing the whole of the properties of the dataset using the COMSA language is done expressing only 32.79% of the number of properties expressed in the Compose dataset. Seen from another angle, we obtain a gain of 3.05, meaning that looking at the Compose properties of the dataset as one set, it is expressed using more that

tree time the properties used to express an at least equivalent set of properties using the COMSA language for description. The average gain is quasi identical with a value of 3.06. The average proportion of properties expressed for a particular application description with the COMSA language relatively to the number of properties expressed for that application description using the Compose is slighly higher at 38.09%. The standard deviation for number of COMSA properties relative to number of Compose properties expressed in percentage or gain, are respectively of 14.3 points and 1.3 indicating a considerable to moderate spread. The medians are respectively at 39.34% of the number of Compose property originally used when describing an application using the COMSA language and the median gain is at 2.54. Through those values we deduce that it is reasonable to expect a gain of 2 to 3 when using the COMSA language to describe a microservice application compared to the use of the Compose language and that conversely we can expect for a Compose described application to be expressed using the COMSA language with only 50% to 35% of the number of properties.

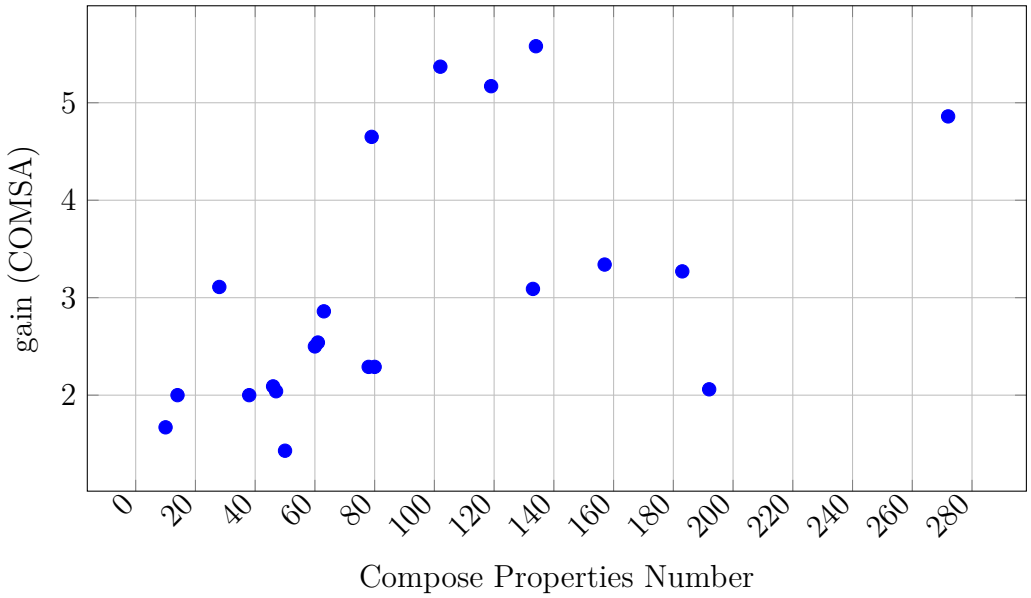


Figure 6.3: Relationship between gain achieved through the use of the COMSA language relatively to the number of properties in the original Compose files.

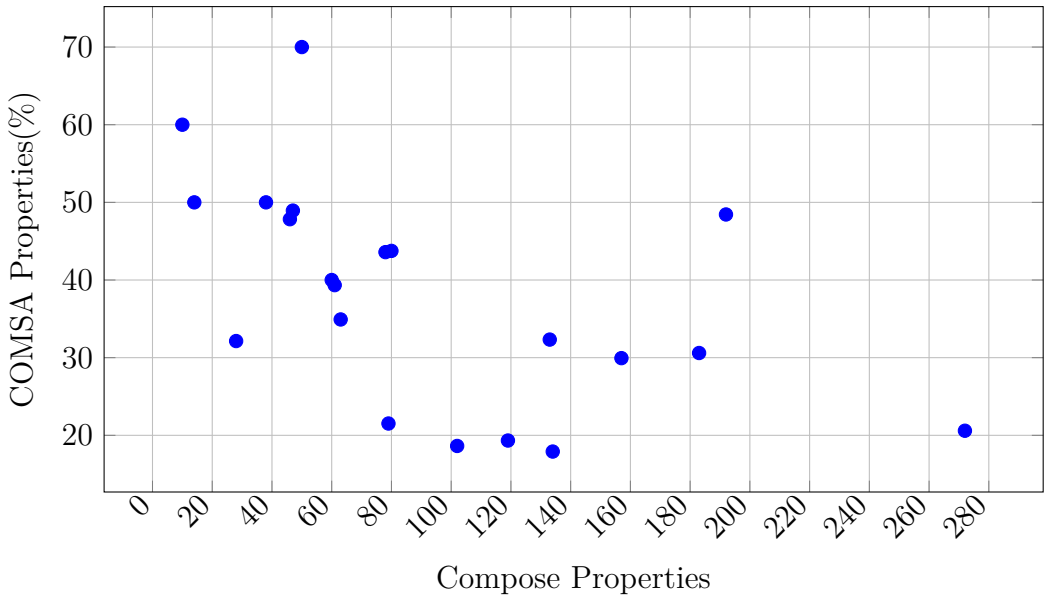


Figure 6.4: Relationship between number of properties in COMSA files (%) relatively to number of properties in original Compose files.

We observe in Table 6.2 an irregular inverse correlation between Compose property number and COMSA property number shown in Figure 6.4 which puts in relation number of Compose properties as the abcissa and the percentage of COMSA properties as the ordinate and symmetrically an irregular correlation between Compose property number and COMSA gain shown in Figure 6.3 which puts in relation number of Compose properties as the abcissa and the gain as the ordinate. That correlation indicates that using the COMSA language on larger applications increases its benefits.

A remarkable feat of the COMSA language is expressed in the "Diff" column that indicates the difference between unique properties and COMSA properties. We note that the values are never positive, meaning that the COMSA language realises the full extent of the possible gain for every application, but that it is often negative. On average the COMSA language uses 5.2 less properties than the unique properties expressed in the original Compose files. That gain is explained by implicit properties and analysed in the next section.

6.3 Dependencies

In Section 5 we made the central distinction in Concerns nature between property centric concerns and service centric concerns. While property centric concerns can be legitimate we consider that whenever possible the architecture description gains from them being integrated in service centric concerns, and particularly into architectural patterns that are by nature significant in describing the architecture. Section B explains

how those patterns can provide the possibility to make some properties implicit as they are by default implied by the pattern use. The COMSA language implements implicit properties through the object oriented pattern Command that allows for modifying the default behavior, thus leaving flexibility to the user while enforcing good practive by design in generating those properties when compiling to an execution language.

At the time of writing, deployment dependencies i.e. `depends_on` in Compose, are the only properties with online impact that the COMSA language implements. The COMSA to Compose compiler from the COMSA toolbox also generates properties indicating post deployment oriented connection and oriented messaging from one microservice to another. Those properties attribute names are respectively `x-connect-to` and `x-sends-to` and take a `Listing` of service identifier as value.

Table 6.3 shows the comparison relative to the mentioned properties between the original Compose files from the dataset and generated Compose files from COMSA files describing the same applications through the `comsa2compose-yaml` compiler. For each application in the dataset the "Original Dependencies" column counts `depends_on` and `links` attributes, the "Generated Dependencies" column counts those same attributes but for the generated files plus the number of added attributes and the "Generated Hidden" counts the non impactful properties and their added total.

Application	Explicit Dependencies		Generated Dependencies			Genrerated Hidden		
	depends_on	links	depends_on	links	added	x-connects-to	x-sends-to	added
blog-microservices	0	0	8	0	8	0	0	0
blogpost	11	17	17	0	-11	0	0	0
consul	1	4	4	0	-1	3	0	3
CQRS	9	0	9		0	2	2	4
hotelreservation	17	0	25	0	8	5	0	5
mediamicroservices	0	0	30	0	30	8	0	8
socialnetwork	19	0	24	0	5	11	0	11
ftgo-application	33	0	37	0	4	3	0	3
graphprocessing	0	10	9	1	0	8	0	8
LakesideMutual	10	0	15	0	5	4	0	4
microcompany	0	0	20	0	20	10	4	14
open5gs	31	0	31	0	0	0	0	0
PetClinic	11	0	15	0	4	12	0	12
PiggyMetrics	8	0	18	0	10	4	0	4
Pitstop	19	0	25	0	6	3	4	7
qbike	1	15	21	0	5	0	0	0
robot-shop	12	0	13	0	1	0	1	1
sockshop	0	0	7	0	7	7	1	8
TapAndEat	0	4	4	0	0	0	0	0
teastore	0	0	8	0	8	4	0	4
train-ticket	1	0	29	0	28	41	0	41
Total	183	50	369	1	137	125	12	137
Min	0	0	4	0	-11	0	0	0
Max	33	17	37	1	30	41	4	41
Average	8.7	2.4	17.6	0.1	6.5	6.0	0.6	6.5
Median	8	0	17	0	5	4	0	4
Standard Deviation	10.2	5.1	9.6	0.2	9.5	8.9	1.2	8.9

Table 6.3: Comparison of expressed dependencies between Compose and COMSA files.

Table 6.3 shows that through compiling COMSA files to Compose files, the resulting files are stripped of `links` properties. That comes from the fact that the `links` attribute is deprecated in newer versions of Compose, we can see that some of them, such as in the `consul` application, were actually expressing dependencies and have been captured by architectural patterns expressed in COMSA and ultimately converted to `depends_on` properties. Others have been simply removed by the COMSA description, such as in the `blogpost` application, which is of no consequence since Docker allow for communication

between containers, which was the function of the `links` attribute.
Even though some `links` attributes are lost in the COMSA description and compilation to Compose, the total of dependency related properties go from 233 in the original files in the dataset to 370 in the generated files, adding 59% more impactful dependencies with no presence of the dependency attribute in the COMSA application description. On average, 6.5 dependencies are added for an application written in COMSA with no presence of the dependency attribute which puts in relation to the average microservice number of 15.4 shown in Table 6.1 is close to an additional dependency for every other microservice.

6.4 Concern Presence

The main proposition of this thesis is a new ADL dedicate to microservice-based application. In the process of its making we applied the concern concept to the application in the presented dataset. As our language is tailored to optimally apply the concern concept we can step aside from this particular implementation and study concern presence in the applications of the dataset. Although separation of concern is to some extent equivocal, we're confident in saying that the concern we identified are there and relevant which permits to extract some data to analyse. This is what this section is dedicated to.

6.4.1 Property Centric Concerns

Property centric concerns, to the difference of architectural patterns, have no powerful architectural meaning by nature. As explained in Chapter 5 it is preferable to integrate their properties into architectural patterns as much as possible as long as their properties are justified to be integrated into them. We tried to follow that aim and thus we consider the data presented in Table 6.4 and Table 6.5 to be relative to standalone property centric concerns that are to be kept separated from other concerns. For some of them such as the `Restart` concern that usually indicates an application wide restart policy the cut is clear. For others, such as the `Image` concern it can be more of a description preference but can also be entirely justified for instance by the usage of a repeating pattern that is a naming concern worth highlighting.

Table 6.4 shows and count property centric concern repartition by application in the dataset. Table 6.5 shows the global values in absolute terms by number of instances of each property centric concern in the dataset (Total) as well as the count of applications having at least one instance of a particular property centric concern (uc) and the corresponding proportional coverage of the dataset (uc%) indicating how much the concern crosscuts the applications in the dataset. Figure 6.5 represents the last value in the form of an histogram for easier reading.

Concern	blog-microservices	blogpost	consul	CQRS	hotelreservation	mediamicroservices	socialnetwork	ftgo-application	graphprocessing	LakesideMutual
Image				2	3	4	4	1	1	1
Build	1	2	1		1			1		1
Environment	1	1		1	3			1	1	
ContainerName				1	3				1	
Hostname					1	1	1			
Restart	1				1	1	1			
Deploy										
DeployResourcesLimitsMemory										
CapDrop										
CapAdd										
ReadOnly										
Tmpfs										
Expose		1			3	4	3			
Command										
EntryPoint					1	1	1			
Volumes					1		1			
Logging					1					
Healthcheck										
SharedNetwork										
PublicPorts	1	1	1	1	1	1	1	1	1	1
Cluster	2		1							2
Total	6	5	3	5	19	12	12	4	4	5

Concern	microcompany	open5gs	PetClinic	PiggyMetrics	Pitstop	qbike	robot-shop	sockshop	TapAndEat	teastore	train-ticket
Image	2	0	1	2	1	1	1	2	1	1	1
Build	0	0	0	0	1	0	1	0	0	0	1
Environment	0	0	0	1	2	0	0	1	0	0	0
ContainerName	0	0	1	0	1	0	0	0	0	0	0
Hostname	0	1	0	0	0	0	0	1	0	0	1
Restart	0	1	0	1	0	0	0	1	0	0	1
Deploy	0	0	0	0	0	0	0	0	0	0	1
DeployResourcesLimitsMemory	0	0	2	0	0	0	0	0	0	0	0
CapDrop	0	0	0	0	0	0	0	1	0	0	0
CapAdd	0	0	0	0	0	0	0	3	0	0	0
ReadOnly	0	0	0	0	0	0	0	1	0	0	0
Tmpfs	0	0	0	0	0	0	0	1	0	0	0
Expose	0	0	0	0	0	0	1	2	0	0	0
Command	1	0	0	0	0	0	0	0	0	0	0
EntryPoint	0	0	0	0	0	0	0	0	0	0	0
Volumes	0	0	0	0	0	0	0	0	0	0	0
Logging	0	0	0	1	0	0	1	0	0	0	0
Healthcheck	0	0	0	0	0	0	1	0	0	0	0
SharedNetwork	0	0	0	0	0	0	1	0	0	0	1
PublicPorts	1	1	1	1	1	1	1	1	1	1	1
Cluster	0	1	2	0	0	0	0	0	0	0	1
Total	4	4	7	6	6	2	7	14	2	2	8

Table 6.4: Property centered concerns distribution in the dataset by application.

Concern	Total	uc	uc(%)
PublicPorts	21	21	100
Image	29	17	81
Build	10	9	42.9
Environment	12	9	42.9
Restart	8	8	38.1
Hostname	6	6	28.6
Expose	14	6	28.6
ClusterConcerns	9	6	28.6
ContainerName	7	5	23.8
Logging	3	3	14.3
EntryPoint	3	3	14.3
Volumes	2	2	9.5
SharedNetwork	2	2	9.5
Command	1	1	4.8
CapDrop	1	1	4.8
CapAdd	3	1	4.8
ReadOnly	1	1	4.8
Tmpfs	1	1	4.8
Deploy	1	1	4.8
DeployResourcesLimitsMemory	2	1	4.8
Healthcheck	1	1	4.8

Table 6.5: Property centered concerns distribution in the dataset.

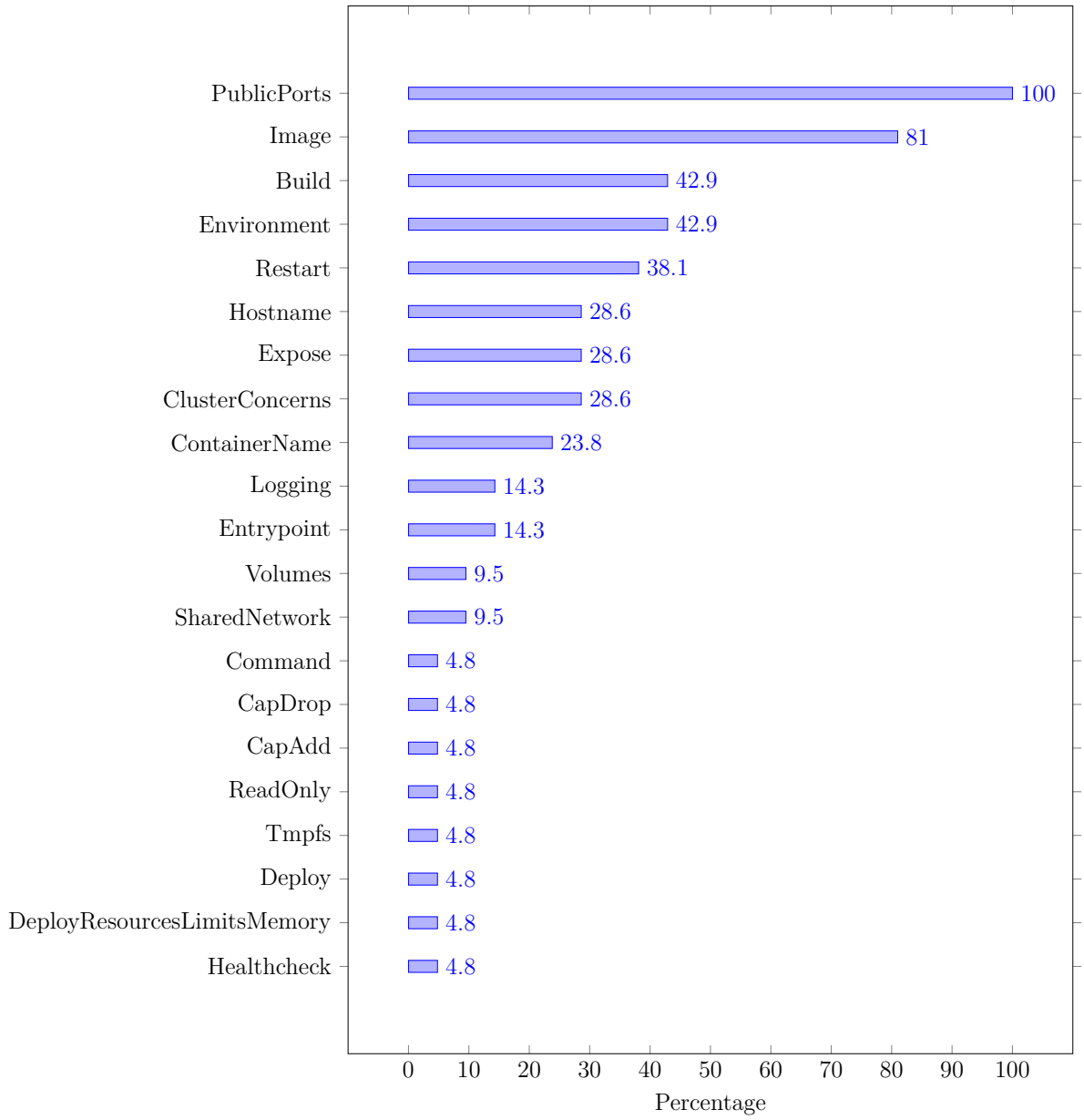


Figure 6.5: Property centric concerns usage proportion in the dataset.

The `PublicPorts` concern has understandably the largest coverage as it is required for any application that is to be interacted with from outside. Exceptions could exist but would be extremely marginal and are not covered in the dataset. The `PublicPorts` is a particular case of property centric concern as it does not aim to unify multiple identical properties but to regroup them in a single place to give an easier view of a critical aspect of the system. The `Image` concern follows closely with 81% dataset coverage which is explained by both the fact that the `Image` attribute is the most common way of providing the application code and that the microservice images often share a naming pattern among an application. The `Build` attribute is a common alternative to the `Image` attribute which provokes the creation of the image at system launch. It can also be used in conjunction to the `Image` attribute to specify the resulting image name which explains

that the sum of its 42% coverage and the 81% coverage of the `Image` is higher that 100%. The fact that close to half the dataset is covered by the `Build` concern indicates, as with the `Image` concern, the commonality of naming patterns in its related attribute usage. The `Environment` concern covers 42% of the dataset, indicating that in close to half the cases, at least part of the environment variables that were not associable with an architectural pattern can be unified in common properties shared between multiple microservices. The `Restart` concern covers 38.1% of the dataset, indicating a restart policy applied on the system. That figure can should lead to look at what's missing i.e. the 61.9% applications that don't have restart policies and thus no explicit plan for service failure. The `Hostname` concern covers 28.6% of the dataset indicating that in more than one out of four applications, it is considered useful by the developers to rely on an explicit name for containers instead of a default one and that when doing so a naming pattern is used. The `Expose` concern covers 28.6% of the application indicating that when documenting the exposed port inside the applications is common for developers to do so in a systematic manner among microservices. The `Cluster` concern covers 28.6% of the applications indicating that in more than one out of four applications we were able to identify groups of heterogeneous properties that were shared by multiple services, indicating an architectural intent, but that we couldn't link to an architectural pattern. That figures indicates that even through analysis, without the proper syntactic tools to reveal the architecture, part of it remains obscure to the external reader even if noticeable. The `ContainerName` concern covers 23.8% of the dataset and indicates that similarly as with `Hostname` concern in around one out of four applications a naming pattern was used to set an explicit name to the container and not rely on the default one. The `Logging` concern covers only 14.3% of the dataset, indicating in contrast that centralized logging is a largely underused policy.

6.4.2 Architecture Pattern Concerns

Architecture pattern concerns are architecturally significant by nature are as they describe the architecture not only in a comprehensive conceptual manner but also from a communicational point of view which allows to augment the determination of services by automatically affecting properties induced by the pattern as shown previously with dependencies.

Table 6.6 shows and counts architectural concern repartition by application in the dataset. Table 6.7 presents for each architectural concern the count of their total occurrences and how much they cross-cut the dataset through the count of applications they are present in and the percentage of applications in the dataset it represents. That last figure is also represented in Figure 6.5 in the form of an ordered histogram.

Concern	blog-microservices	blogpost	consul	CQRS	hotelreservation	mediamicroservices	socialnetwork	ftgo-application	graphprocessing	LakesideMutual
SharedDatabase								2	2	
DatabasePerService				1	9	18	13			
CQRS				1						
ExternalConfiguration	1	1		1					1	
AccessToken	1									
DistributedTracing	1				1	1	1	1		
LogAggregation										
ApplicationMetrics	1	1							1	1
TransactionalOutbox								1		
Messaging	1	1		1				1	1	
CircuitBreaker		1								
ServiceRegistry	1	1	1	1	1			1	1	
ApiGateway	1	1	1	1		1	2	1		
Proxy						1				
Aggregator		1								
SharedService										4
Total	7	7	2	6	11	21	16	7	6	5

Concern	microcompany	open5gs	PetClinic	PiggyMetrics	Pitstop	qbike	robot-shop	sockshop	TapAndEat	teastore	train-ticket
SharedDatabase		1			1	2	3				
DatabasePerService				4				4		1	25
CQRS	2										
ExternalConfiguration	1		1	1					1		
AccessToken	1			1							
DistributedTracing			1			1					
LogAggregation					1						
ApplicationMetrics	1		3	1							
TransactionalOutbox											
Messaging	1			1	2	1	1	1			1
CircuitBreaker	1			1							
ServiceRegistry	1		1	1		1				1	
ApiGateway	1		1	1	1	1	1	1		1	1
Proxy								1			
Aggregator		1									
SharedService		2								1	
Total	9	4	7	11	5	6	5	7	1	4	27

Table 6.6: Architectural concerns distribution in the dataset by application.

Concern	Total	uc	uc(%)
ApiGateway	17	16	76.2
Messaging	13	12	57.1
ServiceRegistry	12	12	57.1
DatabasePerService	75	8	38.1
ExternalConfiguration	8	8	38.1
DistributedTracing	7	7	33.3
ApplicationMetrics	9	7	33.3
SharedDatabase	11	6	28.6
SharedService	7	3	14.3
AccessToken	3	3	14.3
CircuitBreaker	3	3	14.3
CQRS	3	2	9.5
LogAggregation	1	1	4.8
TransactionalOutbox	1	1	4.8

Table 6.7: Architectural concerns distribution in the dataset.

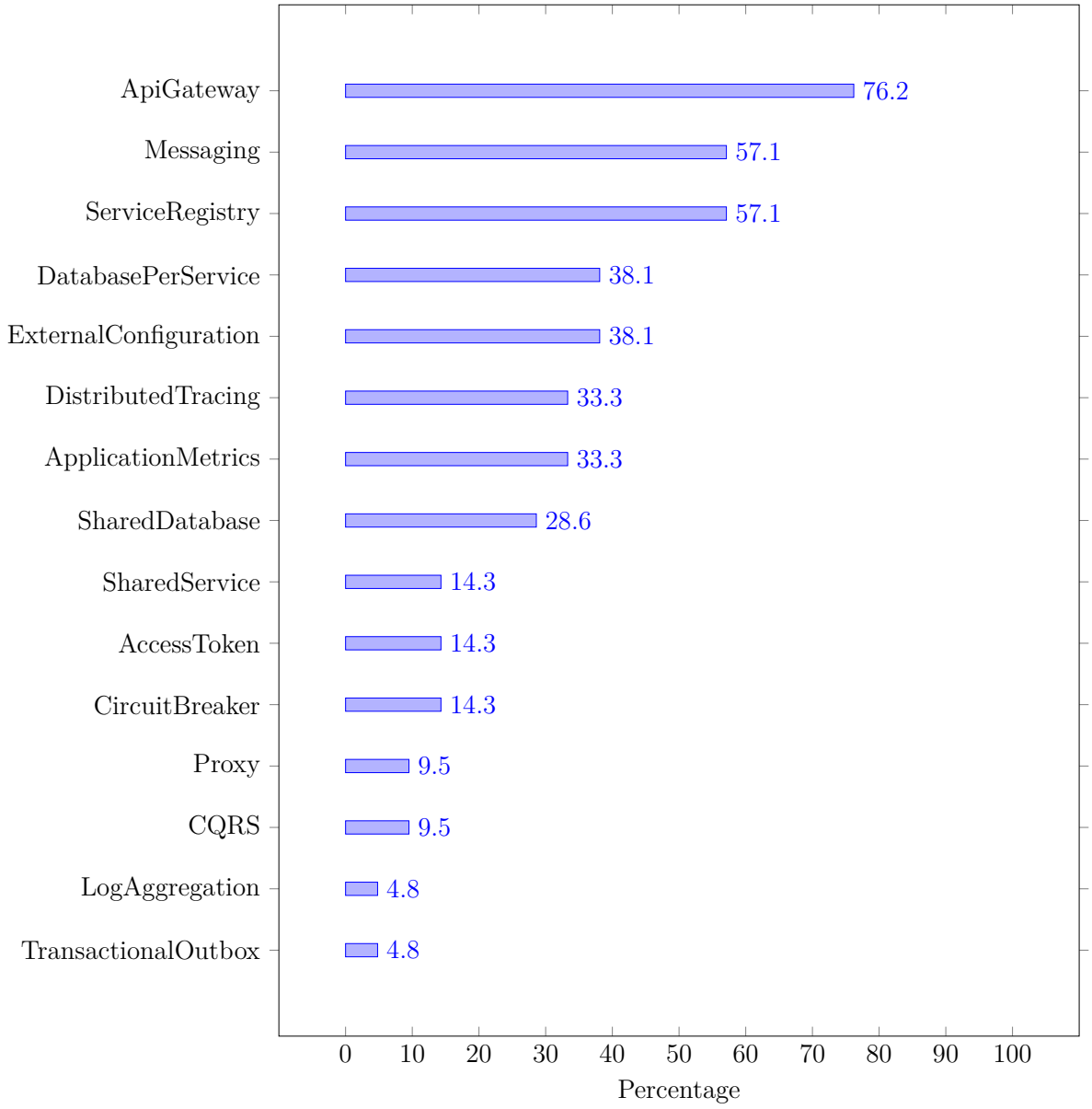


Figure 6.6: Architecture concerns coverage on the dataset.

The most present pattern is the `ApiGateway` concern, which indicates the will of provinding a single, or at least a privileged way to access the application. That figure is to be put in relation with the aforementioned 48.15% of microservices having a port open for external connection hinting to architectural contradictions. The `Messaging` and `ServiceRegistry` patterns come next being both present in 57.1% of the applications and cover in conjunction 76.1% of the dataset. Those two patterns can be thought of as exclusive alternatives for communication but surprisingly their intersection is equal to their cumulated difference with 38% of the application implementing both patterns and the same proportion implementing strictly one of them.

The `DatabasePerService` pattern is canonically considered as the desirable way for storing permanent data but is only used in 38.1% of the applications showing

at the same time the will to apply good practice but also the concrete limitations of distributed data storage. Its distribution in the dataset is strictly distinct to the 28% of applications implementing the `SharedDatabase` pattern indicating application wide architectural commitment toward one or the other pattern. Adding applications implementing the `CQRS` pattern we identified 71.4% of the applications data access and storage architectural types thus leaving more than a quarter of applications either using other types of data management, being entirely stateless or having an original description unclear enough that we were not able to identify it base on their Docker Compose description. External-configuration has a high repartition in the dataset with 38.1% of the applications containing an instance of the pattern, indicating the will to keep microservices as stateless as possible when deployed which is desirable. The observability patterns `DistributedTracing` and `ApplicationMetrics` each cover one third of the dataset indicating both commonality but not general adoption with the majority of developers preferring direct observation than integrating established observational tools. The `SharedService` pattern only reaches 14.3% of the applications indicating that spotted but unidentified patterns are not frequent which puts in relation with the limited number of microservices being part of the property centric concern `Environment` and `Cluster` indicated that service with leftover properties non assignable to legitimate concerns are a relatively small minority.

Architectural patterns can be used to characterize applications at an high level of abstraction independently of its business function. Considered independently they describe particular aspects of the applications functioning but considered in conjunction they provide an immediate understanding of the architecture. For instance, without any information about business aim, knowing that an application has a single entrypoint for client through the implementation of the Api Gateway pattern, using the service registry pattern for communication and a shared database provides an immediate high level picture of the application and outlines the architectural differences with another application that would have multiple client entrypoints, with asynchronous communication through the implementation of the messaging pattern and distributed data access and storage with application wide implementation of the database per service pattern.

`ApiGateway` pattern being the most spread architectural pattern and because of its architecturally determining nature, we chose to use it as an anchor for the forthcoming architectural characterization. Tables 6.8, 6.9 and 6.10 show dataset coverage for sets of pattern meaning that the value indicated in percentage represents the proportion of applications implementing at least one instance of each of the associated patterns.

First Concern	Second Concern	Dataset Coverage (%)
ApiGateway	ServiceRegistry	47.6
ApiGateway	Messenging	52.4

Table 6.8: Dataset coverage of `ApiGateway` coupled with communication patterns.

Table 6.8 shows that the communicational pattern is the second most determining architectural characterizing factor with dataset coverage for `Messaging` pattern and `ServiceRegistry` pattern in conjunction with `ApiGateway` pattern of respectively 52.4% and 47.6%. The conjunction of communicational patterns covers 87.5% of the applications implementing the `ApiGateway` pattern indicating that it is almost implied to use at least one of the two. Among those applications we note that the coexistence of communicational patterns is of the same proportion as in the general case with half the application implementing both patterns and half implementing only one of them.

First Concern	Second Concern	Dataset Coverage (%)
ApiGateway	DatabasePerService	33.3
ApiGateway	SharedDatabase	19.0

Table 6.9: Dataset coverage of `ApiGateway` coupled with databases patterns.

Table 6.9 shows dataset coverage of `ApiGateway` pattern coupled with one of the two most common data access and storage patterns, `DatabasePerService` and `SharedDatabase`, implemented by respectively 33.3% and 19% of the application, thus encompassing together more than half of the dataset. We identify from those figures that those couples of patterns are common fundamental architectural choices.

First Concern	Second Concern	Third Concern	Dataset Coverage (%)
ApiGateway	ServiceRegistry	DatabasePerService	14.3
ApiGateway	ServiceRegistry	SharedDatabase	9.5
ApiGateway	Messenging	DatabasePerService	19.0
ApiGateway	Messenging	SharedDatabase	19.0

Table 6.10: Dataset coverage of `APIGateway` coupled with communication patterns and data access and storage patterns.

Table 6.10 shows the dataset coverage for the two architectural alternatives we identified as prominent i.e. an implementation of `ApiGateway` pattern with either `ServiceRegistry` pattern or `Messaging` pattern and either `DatabasePerService` pattern or `Messaging` pattern. The totality of those combinations covers 38.1% of the dataset, indicating the relevance of those architectural characterization. We note that the implementation of `Messaging` pattern is more commonly associated with at least one data access and storage pattern.

6.5 Concern Scattering

Concern Scattering represents the proportion of microservices pertaining to a Concern in an application. Through it we can assess how much a concern affects an application and consequently how much the use of a concern oriented description language can improve the application description compared to a language using the classical model.

6.5.1 Property Centric Concerns

Table 6.11 shows for each application the percentage of microservice pertaining to a particular concern type. If a concern is not used in the application description the corresponding cell is left empty. Table 6.12 shows the average scattering values for each concern type present multiple times in the dataset. The applications that don't have an instance of the concern type are not included in the average calculation.

Concern	blog-microservices	blogpost	consul	CQRS	hotelreservation	mediamicroservices	socialnetwork	ftgo-application	graphprocessing	LakesideMutual
Image				71	79	91	89	69	67	100
Build	91	67	80		42			69		100
Environment	91	89		57	74			38	22	
ContainerName				71	89				67	
Hostname					26	97	96			
Restart	100				100	94	100			
Deploy										
DeployResourcesLimitsMemory										
CapDrop										
CapAdd										
ReadOnly										
Tmpfs										
Expose	33				84	91	48			
Command										
Entrypoint					42	36	41			
Volumes					32		41			
Logging					16					
Healthcheck										
SharedNetwork										
PublicPorts	64	67	40	100	16		15	88	100	100
Cluster	100		60							67

Concern	microcompany	open5gs	PetClinic	PiggyMetrics	Pitstop	qbike	robot-shop	sockshop	TapAndEat	teastore	train-ticket
Image	91		70	93	69	50	75	93	100	100	62
Build					69		67				62
Environment				57	69			20			
ContainerName			100		77						
Hostname		100						93			100
Restart		93		100				93			62
Deploy											4
DeployResourcesLimitsMemory			100								
CapDrop								93			
CapAdd								80			
ReadOnly								93			
Tmpfs								47			
Expose							50	67			
Command	82										
Entrypoint											
Volumes											
Logging				100			100				
Healthcheck							58				
SharedNetwork							100				100
PublicPorts	100	21	100	36	62	100				29	63
Cluster		86	90								35

Table 6.11: Property centric concerns scattering by applications in the dataset.

Concern	Average
Restart	92.74
Hostname	85.39
ContainerName	80.80
Image	80.53
Cluster	73.00
Logging	72.00
Build	71.85
Expose	62.17
Environment	57.43
PublicPorts	52.42
Entrypoint	39.67
Volumes	36.50

Table 6.12: Property centric concerns scattering average values by concern.

The `Restart` concern is usually meant to be a global policy which explains its high average scattering of 92.74%. The reason for which it does not reach 100% is sometimes explained by a legitimate exception but most of the time it comes from an error of description that is due to the need in languages of the classical model to repeat for each service an identical property. The probability of error is then proportional to the size of the application and then lessen in the same order of magnitude when using a concern oriented ADL alongside induced redundancy reduction and coherence increase. The `Logging` policy is analogous to the `Restart` policy with a lower average scattering value of 72% explainable by some applications storing only logs of specific microservices or to the contrary not storing logs of microservices whose purpose is purely functional. The concerns belonging to the naming pattern overarching concern are by nature highly scattered across applications with values of 85.39% for `Hostname`, 80.80% for `ContainerName`, 80.53% for `Image` and 71.85% for the `Build` concern. The `Cluster` and `Environment` concerns have relatively high scattering with respective values of 73% and 57.43% indicating the unclearness induced by the classical description model since even through analysis we were not able to link the contained properties to explicit architectural structures even though they encompass large portions of the applications and thus must contain architectural significance for the original architect. The `Expose` and `PublicPort` concern are at respectively 62.17% and 52.42% average scattering. The differ from the previously mentioned values since their main purpose is not to provide coherence but to gather sensitive and useful information in a single place. The `PublicPort` value is surprisingly high, indicating that on average half of the microservices are reachable from outside the application. On the other hand the `Expose` value is surprisingly low, indicating that applications descriptions documenting the microservices' reachable ports for other

microservices in the application don't do so in a coherent way, leaving more than a third of microservices undocumented although they necessarily have reachable ports.

6.5.2 Architecture Pattern Concerns

Table 6.13 shows for each application the percentage of microservice pertaining to a particular concern type. If a concern is not used in the application description the corresponding cell is left empty. Table 6.14 shows the average scattering values for each concern type present multiple times in the dataset. The applications that don't have an instance of the concern type are not included in the average calculation.

Concern	blog-microservices	blogpost	consul	CQRS	hotelreservation	mediamicroservices	socialnetwork	ftgo-application	graphprocessing	LakesideMutual
ApiGateway	18.18	44.44	80.0	42.86		24.24	40.74	25.0		
DatabasePerService				28.57	78.95	84.85	74.07			
DistributedTracing	54.55				47.37	39.39	44.44	25.0		
Aggregator		33.33								
SharedService										88.89
SharedDatabase								68.75	44.44	
Messaging	45.45	33.33		42.86				62.5	22.22	
CircuitBreaker		55.56								
ExternalizedConfiguration	81.82	88.89		57.14					44.44	
ApplicationMetrics	54.55	22.22							100.0	55.56
ServiceRegistry	63.64	77.78	100.0	57.14	47.37			62.5	33.33	
AccessToken	54.55									
ProxyServer						6.06				
TransactionalOutbox								50.0		
CQRS				42.86						
LogAggregation										

Concern	microcompany	open5gs	PetClinic	PiggyMetrics	Pitstop	qbike	robot-shop	sockshop	TapAndEat	teastore	train-ticket
ApiGateway	54.55		40.0	35.71	30.77	50.0	41.67	46.67		71.43	61.76
DatabasePerService				57.14				53.33		28.57	73.53
DistributedTracing			50.0			50.0					
Aggregator		42.86									
SharedService		78.57								57.14	
SharedDatabase		42.86			53.85	70.0	66.67				
Messaging	54.55			28.57	76.92	70.0	25.0	20.0			7.35
CircuitBreaker	54.55			42.86							
ExternalizedConfiguration	90.91		70.0	64.29					100.0		
ApplicationMetrics	54.55		70.0	42.86							
ServiceRegistry	63.64		60.0	50.0		60.0				85.71	
AccessToken	18.18			14.29							
ProxyServer								13.33			
TransactionalOutbox											
CQRS	45.45										
LogAggregation					76.92						

Table 6.13: Architectural concerns scattering by application.

Concern	Average
LogAggregation	76.92
SharedService	74.87
ExternalizedConfiguration	74.69
ServiceRegistry	63.43
DatabasePerService	59.88
SharedDatabase	57.76
ApplicationMetrics	57.1
CircuitBreaker	50.99
TransactionalOutbox	50.0
DistributedTracing	44.39
ApiGateway	44.25
CQRS	44.16
Messaging	40.73
Aggregator	38.1
AccessToken	29.0
ProxyServer	9.7

Table 6.14: Architectural concerns scattering average values by concerns.

The `ExternalConfiguration` pattern has an average scattering value of 76.92% indicating that when used it is implemented for the majority of services in the application leaving some specific microservices with either local or no particular configuration. The `DatabasePerService` reaches an average scattering value of 59.88% indicating that although it is by nature a local pattern it is, when used, widely implemented through the application leaving around 40% of microservices out of it which is explained not by architectural inconsistency but simply by the absence of a need for a database. This explanation is comforted by the approximatively same values for the `SharedDatabase` pattern. This correlation tends to indicate that the proportion of microservices needing access to a database is on average between 50% and 60%. The `CQRS` pattern which is another form of shared databases reinforces that analysis with an average scattering of 44.16%. Another deduction can be made about the `ServiceRegistry` pattern that reaches on average only 63.43% of the services in an application while it is meant to be the central communication mechanism. That implies that a consequential third of the microservices are not relying on the pattern to communicate either having communicational data hard coded or using another centralized communication pattern. The `Messaging` pattern is an alternative that is shown not to integrate the majority of microservices in the applications with an average scattering of 40.73% and thus largely being used as a secondary mean of communication. The observational patterns, `ApplicationMetrics` and `DistributedTracing` have average scattering values of respectively 57.1% and 44.39% indicating that metrics observation is

considered significant for a larger number of microservices than communication tracing is, from which we can infer that the application core is relatively restricted. Ultimately the `SharedService` pattern is scattered on average through 74.87% of the microservices. Since that pattern is a construction meant to fill an identified communication relation for which we were not able to determine a precise architectural function indicating the necessity of a language that implements explicit architectural structures by design.

6.6 Service Tangling

Service Tangling represents how much a service is part of the different patterns in a particular application. Because of that, and to the difference of scattering, it cannot be directly generalised to a dataset of applications. Instead, and to keep the spirit of what tangling measures, we decide to look at how much services are part of each concern types.

6.6.1 All Services

Table 6.15 shows the property centric concern coverage but to the difference of Table 6.5 it shows the proportion of microservices in the dataset covered by concerns instead of the applications.

Concern	%
Image	68.21
Restart	52.78
Hostname	49.07
PublicPorts	48.15
Build	32.72
SharedNetwork	24.69
Expose	24.07
Environment	19.75
ContainerName	14.81
Entrypoint	9.57
Logging	8.95
Cluster	8.95
Volumes	5.25
CapDrop	4.32
ReadOnly	4.32
Deploy	4.01
CapAdd	3.7
Command	2.78
Healthcheck	2.16
Tmpfs	2.16

Table 6.15: Property centered concerns microservice coverage.

Unsurprisingly the `PublicPorts` concern is relegated but still reaches almost half (48.15%) of the microservices in the dataset indicating that being opened to the exterior is common which possibly implies unnecessary security risks. The `Image` concern is still prevalent with 68.21% of microservices contained compared to the 32.72% of the `Build` concern alternative. The `Environment` and `Cluster` concerns encompass respectively only 19.75% and 8.95% of the microservices indicating a limited part of remaining configuration properties that while being shared were not identified as part of a pattern contrary to their application presence of 42.9% and 28.6%.

However looking at the proportion of microservices participating in architectural concerns makes less sense. The infrastructure microservices are by definition part of the concern they are at the center of which makes the data unclear, also they are less subject to be at the crossroad of other architecture concerns and most importantly there is no obvious meaning that we can expect from that analysis. On the other hand that approach applied only to business services can sketch a profile for their abstracted average composition through the architectural patterns they take part in.

6.6.2 Business Services

Before looking at the patterns business services are on average entangled in, it is instructive to look at the repartition of business services in the dataset relatively to infrastructure services. We consider business services as those not being at the center of an architecture pattern. We exclude the Shared Service pattern as an appropriate distinction as it can represent unidentified architecture pattern as well as a particular composition of business microservices communication. Figure 6.7 puts in relation the proportion of business services in the application as the absissa and as the ordinate the total number of microservices.

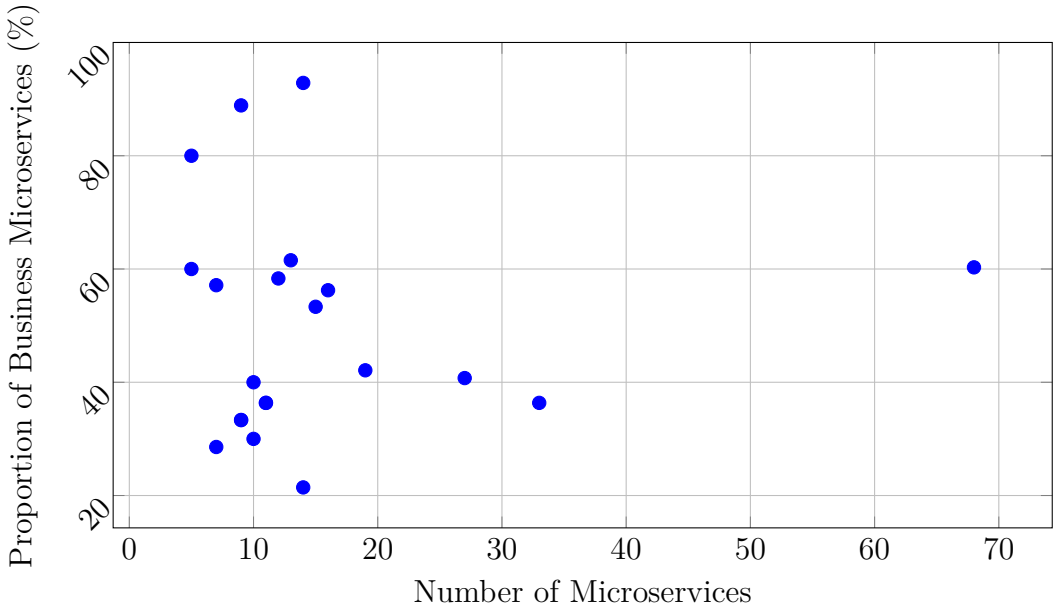


Figure 6.7: Relationship between proportion of business services (%) and number of microservices.

No clear relation can be extracted from Figure 6.7 implying that the proportion of business services is not dependent of the size of the application but rather of the implementation choices of the architect. We can still extract the average proportion of business microservices which in our dataset is of 49.9% thus implying an equal split with infrastructure services.

Table 6.16 shows the coverage proportion of architecture patterns on business microservices in the dataset. From it we can extract that the most common business service profile is constituted above all by being connected to a gateway (61.73%), often not being connected to a database but if so, having its own dedicated database (35.19%) is two times more likely that sharing a database (18.52%). Communication means are split between service discovery through a registry (29.01%) and asynchronous communications through a message broker (25.93%) with a little advantage for the former. Around half the business microservices don't have an explicit communication mechanism thus relying

in the case of the source files on the integrated Docker DNS. Tracing (26.54%) and metrics (12.96%) are only applied to a considerable minority.

Concern	%
ApiGatewayPattern	61.73
DatabasePerServicePattern	35.19
ServiceRegistryPattern	29.01
DistributedTracingPattern	26.54
MessagingPattern	25.93
SharedDatabasePattern	18.52
ExternalizedConfigurationPattern	15.43
ApplicationMetricsPattern	12.96
CircuitBreakerPattern	6.17
LogAggregationPattern	4.94
TransactionalOutboxPattern	4.32
CQRS_Pattern	3.7
AccessTokenPattern	2.47

Table 6.16: Architecture centered concerns business microservice coverage.

6.7 Conclusion

In this chapter we have established the gain brought by the use of a concern-based language to describe the architecture of microservice-based applications. Applying the COMSA language on a dataset of 21 applications written in Compose we needed on average less that one-third of the number of properties to express an at least equivalent description of an application. In all cases a gain was observed and tended to be proportional to the number of properties in the original description. This gain was reinforced by the ability of a concern-based language to make certain properties implicit. This result directly shows a gain in coherence as the use of a concern-based language allows to unify identical properties meant to be unique properties in single places dedicated to the concern the property pertains to. From it we can expect harder to measure gains such as increased understandability and reduced error proneness.

Having established a dataset of applications written by applying the concern concept we went on to analyse concern presence independently from our particular implementation using our distinction between property centric concern, meant to express the application's policies, and architecture pattern concerns, meant to express the application's topologies. About property service concerns, we noticed that less than 40% of the applications had a restart policy, that 42% of the application had multiple services sharing the same environment properties and that for more than a fourth of the applications we were able

to identify clusters of properties shared by multiple services without being able to tie them to an identified concern. About architecture pattern concerns, we noted the prevalence of the api gateway concern reaching more than tree quarters of the applications, the equal split in communication pattern between messaging and registry both reaching more than a half of application thus having a notable intersection and cumulative extension of data access and storage patterns to 71% of microservices. From those considerations on architecture pattern concerns repartition we propose the idea of a modular architectural characterization of applications.

We analysed concern scattering among microservices in applications and observed that restart policies had an average scattering of 92% hinting toward oversights due to language limitations. We also noted that more than on average more than half of the microservices had open ports hinting toward security deficiencies. The high average scatterings of the cluster concern and the shared service concern indicate fundamental architectural unclearness in the original descriptions.

While the analysis of concern presence provide insights about the underlying architectural structures used to model microservice-base applications, it indicated in many cases error-proneness or obscurity in the description that reinforces our claim for the benefits brought by using a concern-based language for microservice architecture description.

Chapter 7

Conclusion and Perspectives

7.1 Conclusion

The microservice style has delivered most of its anticipated gains, becoming through its merits an industry standard. Implementing an application with a microservice architecture brings unique benefits on many levels among which scalability, fault tolerance, technology freedom and continuous integration and deployment coming from independence of development teams. All those radical improvements find their origin in a technical aspect which is the independence of execution of microservices each being a distinct process whose surface of contact with the other parts of the application is defined and limited to a clear contract, making, from an outside viewpoint, the inner workings a black box.
However those appealing aspects of microservices come at a cost. The segmentation of traditional monolithic programs into multiple smaller or even capability atomic programs largely reduces or removes the risk of uncontrolled function coupling. However it transfers the risk of architectural disarray to microservice organisation with the risk of a self reinforcing negative loop of unnecessary coupling, unintelligible architecture, proliferation of entities and unnecessary function calls.
Because no definitive method or tool has resolved this difficulty, architect must rely on thoughtful prevision at design time in order to anticipate future architectural risks since the one of the major benefits of microservice architecture is to limit drastically the need for centralized decision. Indeed, one of the constraints of monolithic architecture was the need of centralized or at least extensive coordination between teams working on different parts of the same program, reducing that necessity to the minimum the microservice style has made the evolutional control largely rely on developers decisions. To decide on local decision while having the consequences on the entire system in mind it is necessary to first dispose of a simple view of the application. Architecture Description Languages dedicated to microservice applications are meant to provide that global abstraction.
The current landscape of microservice ADL is composed of solutions largely sharing the same underlying model where descriptions consist of a list of microservices focusing more

on the component composition than on the application topology. From that realization we proposed in this thesis a new model extending the one we observed in other microservice ADLs by adding new structure as first class citizen representing the meaningful aspects of the architecture. With this perspective in mind, we used the Concern concept to give its semantic to the structure, which has been the subject of a long reflection in informatics and that aims to group and separate elements in function of architectural appartenance. To achieve that aim, the structure we propose associates sets of microservices with sets of properties pertaining to a particular interest effectively removing concern scattering between microservice descriptions. The corollary is a shift toward coherence happening by unifying properties meant to be unique but that would appear as distinct identical redundant properties in the already existing model. One historical difficulty of Concerns in programming is the non obvious way of merging the elements they contain as part of lower level, concretely executable, elements. That problem being linked to the importance of the order of statements in procedural programming we decided that our model was only relevant in a declarative context.

Having established a new model for microservice application architecture description, we went on to develop the Concern Oriented Microservice Architecture language out of it by developing a concrete syntax. We used Pkl, a new programming language oriented toward configuration to determine file structure and define objects representing the concerns specific to microservice architecture. The central elements of our language are the Concerns implemented as a tree of classes whose root is the Concern class. To make our language useful, we established a library of reusable concerns represented as children classes of the Concern parent class. Besides the principal benefit of expressing clearly the architectural construct, specific Concerns allow to remove explicit expression of induced properties, lightening the description. Establishing that hierarchy brought us to further our reflection and propose a new dichotomy of architectural entities having on one hand concerns centered on a property and on the other concerns centered on a service. We then realized that technical services could be distinguished from business services in a syntactic way, realizing that they were by nature at the center of a pattern.

We then developed a toolbox to support the COMSA language and first and foremost compilers targeting executable languages. We decided on Compose and Kubernetes, both declarative deployment languages, to avoid the flaw previously mentioned of other microservice ADLs whose compilers produce procedural business code needing modification and restricting technology freedom. We ensured that any description made in COMSA would be compiled to an equivalent description in target languages and be immediately executable. That way COMSA can be considered an executable language and a viable replacement for any deployment language for which a compiler has been developed. Visualisation is another asset of our toolbox with one representing Concern microservice extension, allowing to get a quick representation of the system and spot inconsistencies, another describing the application deployment plan and online connections based on

the patterns expressed in the textual description. Finally, we propose a set of analytic tools providing metrics, error checking, and architectural informations about COMSA application descriptions.
While we were confident in the gains provided by the COMSA language, we needed to measure it on real world applications. To that effect, we produced COMSA descriptions of 21 applications from a curated dataset using their Docker Compose description. The first metric we looked at is the number of properties needed for expressing applications descriptions in COMSA that were at least equivalent to their original files. We found that for all applications using COMSA provided substantial gain, with an average of tree times less properties contained in the COMSA description. That result directly confirms the coherence gain and hints strongly on consequential qualitative gains namely maintainability, error resilience and most importantly understandability. We used that dataset of COMSA expressed applications to extract insight relative to spreading of Concerns in microservice applications. For instance, we observed the limited application of restart policies or the wide usage of the API gateway pattern. We propose a characterization of application through sets of implemented patterns showing the widely spread presence of a modular API gateway application parametrized by the communication type and the data storage and access mode.
In summary, this thesis proposes a new paradigm to describe the architecture of microservice application, whose semantic revolves around the Concern concept to represent as first class citizens the chosen structures and intents of the architects, for which a concrete syntax resulting in COMSA, a new executable deployment language was developed. The language is completed by a library of reusable Concerns and an extensible toolbox making it an immediate and viable alternative to existing solutions. The benefits this new approach have been shown by the substantial gain in needed properties to express equivalent architectures indicating substantial advantages in terms of coherence, understandability, maintainability and error resilience. This result has been established through the creation of a dataset of applications descriptions in COMSA which consequently permitted an analysis of real world microservice applications relatively to Concerns such as policies and design pattern.

7.2 Perspectives

In this thesis, we presented COMSA a new language to describe the architecture of microservices applications. Some improvements and extensions could be made but we are confident in affirming that at time of writing the language is immediatly usable with solid theoritical foundations, a coherent concrete syntax, an extensive and easily expandable concern library and compilers not only adapted to its current state but able to process any extensions users might find relevant. Less fundamental but useful tools have been developped and we find that most of the future improvements should revolve around the

toolbox extension. We categorized this chapter by research orientations, indicating the perspective they offer in tool realisation.

7.2.1 Architecture Reconstruction

We spent some time trying to develop an offline tool to extract the structure of applications described in Compose to automatically propose a Concern based description which would obviously be a detemining factor to obtain adoption of the COMSA language. Unfortunately our efforts were not successful and we didn't manage to propose a viable solution. The major difficulty was that a same pattern, identifiable through human analysis, can emerge from very different sets of properties. We were not able to extract enough generality for complex patterns in their related properties accross applications to be able to detect them in application the detecting algorithm had no prior knowledge of. Online analysis is more direct in that regard but unsatifying in our view as we aimed at producing a quasi immediate proposition without technical restrictions such has having the infrastructure to run large microservice systems. Although we were disapointed we still think that this tool is realisable and that its success relies mainly on dataset size, and analysis of the properties constituting specific concerns. Given the time, we would also try AI categorization algorithms.

7.2.2 Behavior, analysis and evaluation

We touched on Concern behavior and its implications on compilation, architecture analysis and error warning with two different aspects of the language: behaviors and constrains. Those two aspects have the same source, the identification of the specific essence of Concerns which permits to derive execution properties while keeping them implicit in the description, suggest improvements to the architect and warn of potential architectural defects. Concern-based architecture description is limitless in terms of behavior implementation, staying at the heart of the microservice spirit by abstracting architectural structures, not infringing on technological freedom. However behavior implementation require deep understanding of each pattern and careful transcription into the language. Indeed mistakes in this domain are much worse than no implementation. The former has the potential of locking the user in faulty structures while the latter simply relies on user knowledge. Those dangers have been mitigated by creating behaviors as modules of Concerns that are sometimes affected by default but can always be deactivated. We are confident that if done carefully work on this aspect of the language can bring many benefits while keeping its generality and openness to specific systems and future evolutions. Static analysis leading to architecture evaluation, positive suggestions and mistakes corrections are realistic expectations but another major domain of application would be operation management through implication of online rules for orchestrators such as Kubernetes controllers.

7.2.3 Representation

The main pragmatical aim of the creation of the COMSA language is to propose an easy understanding of the system while alleviating design work instead of adding an additional burden of maintaining a comprehensible side documentation that is because of the dynamic nature of the microservice style always behind, to a larger or lesser degree, the actual state of the application. To that effect we set ourselves on reconciling documentation and executable code and decided on deployment and operation code because of its often declarative nature and its holistic view. However the fastest way to get a global high level view of the system is still a graphical representation. Having ensured the equivalence between execution code and textual description, we added tools producing visual representations based on the textual description leaving no possible discrepancy between the two. However much work can still be done to expand the visual representations of Concern based descriptions, in particular by providing dynamic view of the system in different levels of abstractions and by incorporating more relevant behavioral visual components.
KubeDiagrams[67] is remarkable instance of a graphical tool akin to our proposition whose origin is traceable to the work done on the COMSA graphical representation. Developed by Philippe Merle, the director of this thesis, it has reached quick adoption in the cloudnative community because of the relevance and usefulness it brings through graphical kubernetes resources regrouping and organization. Bridging the gap between the high level Concerns of COMSA and relatively low level ressource gathering and hierarchical order of KubeDiagram will require extensive work and careful consideration but promises a long lasting representation of microservice application architectures.

7.2.4 Compilers and Language Extension

A natural addition to the the toolbox would be adding compilers targetting other deployment and operation languages such as Nomad[40]. More generally we chose the attributes of microservices as defined by the Compose specification for our service description as we felt they capture the core of what microservices are from a deployment and operational viewpoint adding little properties that were specific to the underlying execution software. This choice has the major benefit to provide easy access to the user due to the popularity of Compose but may limit the quality of the translation toward other execution languages. For that reason it would be of interest to apply the same logic used to create the COMSA language using different basis for service description, e.g., Kubernetes, or even try for a greater level of generality.

Bibliography

[1] Miltiadis Allamanis, Earl T Barr, Christian Bird, and Charles Sutton. "Learning natural coding conventions". In: *Proceedings of the 22nd acm sigsoft international symposium on foundations of software engineering*. 2014, pp. 281–293.

[2] Alloy Team. *Alloy: Lightweight Modeling and Analysis of Software*. https://alloytools.org/. Accessed: 2025-07-12. 2025.

[3] Gustavo Alonso, Fabio Casati, Harumi Kuno, Vijay Machiraju, Gustavo Alonso, Fabio Casati, Harumi Kuno, and Vijay Machiraju. *Web services*. Springer, 2004.

[4] Kenny Bastani. *spring-boot-graph-processing-example*. https://github.com/kbastani/spring-boot-graph-processing-example. GitHub repository, retrieved on 2025-08-22. 2025.

[5] Antonio Brogi, Davide Neri, and Jacopo Soldani. "Freshening the Air in Microservices: Resolving Architectural Smells via Refactoring". In: *Service-Oriented Computing – ICSOC 2019 Workshops*. Ed. by Sami Yangui, Athman Bouguettaya, Xiao Xue, Noura Faci, Walid Gaaloul, Qi Yu, Zhangbing Zhou, Nathalie Hernandez, and Elisa Y. Nakagawa. Cham: Springer International Publishing, 2020, pp. 17–29. ISBN: 978-3-030-45989-5.

[6] Fernando A. B. Campos. *spring-netflix-oss-microservices*. https://github.com/fernandoabcampos/spring-netflix-oss-microservices. GitHub repository, retrieved on 2025-08-22. 2025.

[7] Joe Cao. *Qbike: Cloud-Native Microservices-based Bike Rental System*. https://github.com/JoeCao/qbike. Accessed: 2025-08-05. 2025.

[8] Tomas Cerny, Amr S. Abdelfattah, Vincent Bushong, Abdullah Al Maruf, and Davide Taibi. "Microservice Architecture Reconstruction and Visualization Techniques: A Review". In: *2022 IEEE International Conference on Service-Oriented System Engineering (SOSE)*. 2022, pp. 39–48. DOI: 10.1109/SOSE55356.2022.00011.

[9] Paul C. Clements. "A Survey of Architecture Description Languages". In: *Proceedings of the 8th International Workshop on Software Specification and Design*. IWSSD '96. USA: IEEE Computer Society, 1996, p. 16. ISBN: 0818673613.

[10] *Cloudify Documentation Center*. https://docs.cloudify.co/. Accessed: 2025-08-21. 2025.

[11] Gartner Peer Community. *Microservices Architecture: Have Engineering Organizations Found Success?* Accessed July 2025. 2024. URL: https://www.gartner.com/peer-community/oneminuteinsights/microservices-architecture-have-engineering-organizations-found-success-u6b.

[12] José María Conejero, Juan Hernández, Elena Jurado, and Klaas van den Berg. "Crosscutting, what is and what is not?: A Formal definition based on a Crosscutting Pattern". In: *target* 1.t2 (2007), t3.

[13] George F Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed systems: concepts and design*. pearson education, 2005.

[14] DeathStarBench Developers. *DeathStarBench: Hotel Reservation Application*. https://github.com/delimitrou/DeathStarBench/tree/master/hotelReservation. Accessed: 2025-08-05. 2025.

[15] DeathStarBench Developers. *DeathStarBench: Media Microservices Application*. https://github.com/delimitrou/DeathStarBench/tree/master/mediaMicroservices. Accessed: 2025-08-05. 2025.

[16] DeathStarBench Developers. *DeathStarBench: Social Network Application*. https://github.com/delimitrou/DeathStarBench/tree/master/socialNetwork. Accessed: 2025-08-05. 2025.

[17] Christina Delimitrou et al. *DeathStarBench: An open-source benchmark suite for cloud microservices*. https://github.com/delimitrou/DeathStarBench. GitHub repository, retrieved on 2025-08-22. 2025.

[18] Microservices Demo. *Sockshop: A Microservice Demo Application*. https://github.com/microservices-demo/microservices-demo. GitHub repository, retrieved on 2025-08-22. 2025.

[19] Descartes Research Group. *TeaStore Docker Compose File (default configuration)*. https://github.com/DescartesResearch/TeaStore/blob/master/examples/docker/docker-compose_default.yaml. Accessed on August 3, 2025. 2025.

[20] Descartes Research Group. *TeaStore: A Microservice Reference Application*. https://github.com/DescartesResearch/TeaStore. Accessed: 2025-07-12. 2017.

[21] Arie van Deursen, Paul Klint, and Joost Visser. "Domain-Specific Languages: An Annotated Bibliography". In: *SIGPLAN Not.* 35.6 (June 2000), pp. 26–36. ISSN: 0362-1340. DOI: 10.1145/352029.352035. URL: https://doi.org/10.1145/352029.352035.

[22] Docker, Inc. *Compose file reference*. 2025. URL: https://docs.docker.com/reference/compose-file/ (visited on 08/02/2025).

[23] Docker, Inc. *Deploy configuration reference: replicas*. https://docs.docker.com/reference/compose-file/deploy/. Accessed: 2025-07-12. 2025.

[24] Docker, Inc. *Docker Engine Networking*. Accessed: 2025-08-05. 2025. URL: https://docs.docker.com/engine/network/.

[25] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, Larisa Safina, and Bertrand Meyer. "Microservices: Yesterday, Today, and Tomorrow". In: *Present and Ulterior Software Engineering*. Cham: Springer International Publishing, 2017, pp. 195–216. ISBN: 978-3-319-67425-4. DOI: 10.1007/978-3-319-67425-4_12. URL: https://doi.org/10.1007/978-3-319-67425-4_12.

[26] Iván Dugalic. *Micro Company*. https://github.com/idugalic/micro-company. GitHub repository, retrieved on 2025-08-22. 2025.

[27] Callista Enterprise. *blog-microservices*. https://github.com/callistaenterprise/blog-microservices. GitHub repository, retrieved on 2025-08-22. 2025.

[28] Thomas Erl. *SOA Principles of Service Design (Paperback)*. 1st. USA: Prentice Hall Press, 2016. ISBN: 0134695518.

[29] Javier Esparza-Peidro, Francesc D. Muñoz-Escoí, and José M. Bernabéu-Aubán. "Modeling microservice architectures". In: *Journal of Systems and Software* 213 (2024), p. 112041. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2024.112041. URL: https://www.sciencedirect.com/science/article/pii/S0164121224000840.

[30] Eric Evans. *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional, 2004.

[31] Julián Ferrater. *Tap-And-Eat-MicroServices*. https://github.com/jferrater/Tap-And-Eat-MicroServices. GitHub repository, retrieved on 2025-08-22. 2025.

[32] Fudan SE Lab. *Train-Ticket: A Benchmark for Microservice Architectures*. https://github.com/FudanSELab/train-ticket. Accessed: 2025-08-05. 2022.

[33] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. "An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems". In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '19. Providence, RI, USA: Association for Computing Machinery, 2019, pp. 3–18. ISBN: 9781450362405. DOI: 10.1145/3297858.3304013. URL: https://doi.org/10.1145/3297858.3304013.

[34] *Github - The Compose Specification*. https://github.com/compose-spec/compose-spec.

[35] Nuno Gonçalves, Diogo Faustino, António Rito Silva, and Manuel Portela. "Monolith Modularization Towards Microservices: Refactoring and Performance Trade-offs". In: *2021 IEEE 18th International Conference on Software Architecture Companion (ICSA-C)*. 2021, pp. 1–8. DOI: 10.1109/ICSA-C52384.2021.00015.

[36] Gradiant. *5G-Images: Containerized Microservices for Benchmarking and Testing*. https://github.com/Gradiant/5g-images. Accessed: 2025-08-05. 2025.

[37] Giona Granchelli, Mario Cardarelli, Paolo Di Francesco, Ivano Malavolta, Ludovico Iovino, and Amleto Di Salle. "Microart: A software architecture recovery tool for maintaining microservice-based systems". In: *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*. IEEE. 2017, pp. 298–302.

[38] IMARC Group. *Global Microservices Architecture Market Witnesses Strong Growth Rate of CAGR 13.4%*. Accessed July 2025. 2024. URL: https://www.einpresswire.com/article/692553652/global-microservices-architecture-market-witnesses-strong-growth-rate-of-cagr-13-4-exceeding-us-11-8-billion-2024-32.

[39] Claudio Guidi, Ivan Lanese, Manuel Mazzara, and Fabrizio Montesi. "Microservices: A Language-Based Approach". In: *Present and Ulterior Software Engineering*. Ed. by Manuel Mazzara and Bertrand Meyer. Cham: Springer International Publishing, 2017, pp. 217–225. ISBN: 978-3-319-67425-4. DOI: 10.1007/978-3-319-67425-4_13. URL: https://doi.org/10.1007/978-3-319-67425-4_13.

[40] HashiCorp. *Nomad Documentation*. https://developer.hashicorp.com/nomad. Official documentation site, retrieved on 2025-08-22. 2025.

[41] Sara Hassan and Rami Bahsoon. "Microservices and Their Design Trade-Offs: A Self-Adaptive Roadmap". In: *2016 IEEE International Conference on Services Computing (SCC)*. 2016, pp. 813–818. DOI: 10.1109/SCC.2016.113.

[42] hmonfleur. *Concern Oriented MicroService Architecture: Language and Toolbox)*. https://github.com/hmonfleur/comsa. GitHub repository, retrieved on 2025-08-22. 2025.

[43] *https://kubernetes.io/*. https://kubernetes.io/fr/.

[44] Fortune Business Insights. *Cloud Microservices Market Size, Share & Growth Report, 2025-2032*. Accessed July 2025. 2025. URL: https://www.fortunebusinessinsights.com/cloud-microservices-market-107793.

[45] Spherical Insights. *Global Cloud Microservices Market Size To Exceed USD 8.45 Billion By 2032*. Accessed July 2025. 2024. URL: https://www.globenewswire.com/news-release/2024/01/23/2814330/0/en/Global-Cloud-Microservices-Market-Size-To-Exceed-USD-8-45-Billion-By-2032-CAGR-of-20-49.html.

[46] Instana Contributors. *Robot Shop: Sample Microservices Application*. https://github.com/instana/robot-shop. Accessed: 2025-08-05. 2023.

[47] Technical Committee : ISO/IEC JTC 1/SC 7 Software and systems engineering. *Software, systems and enterprise — Architecture Description*. Standard. International Organization for Standardization, Nov. 2022.

[48] jaespei. *Komponents*. Version e2728ec. GitHub repository; GPL-3.0 license. Oct. 21, 2024. URL: https://github.com/jaespei/komponents (visited on 08/21/2025).

[49] JetBrains. *IntelliJ IDEA*. https://www.jetbrains.com/idea/. Official product website, retrieved on 2025-08-22. 2025.

[50] *JHipster - Fullstack Platform for the Modern Developer!* https://www.jhipster.tech/.

[51] *JHipster Domain Language*. https://www.jhipster.tech/.

[52] *Jolie Programming Language - Official Website*. https://www.jolie-lang.org/.

[53] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, et al. "Cloud programming simplified: A berkeley view on serverless computing". In: *arXiv preprint arXiv:1902.03383* (2019).

[54] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G Griswold. "An overview of AspectJ". In: *European Conference on Object-Oriented Programming*. Springer. 2001, pp. 327–354.

[55] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. "Aspect-oriented programming". In: *European conference on object-oriented programming*. Springer. 1997, pp. 220–242.

[56] Jóakim von Kistowski, Simon Eismann, Norbert Schmitt, André Bauer, Johannes Grohmann, and Samuel Kounev. "TeaStore: A Micro-Service Reference Application for Benchmarking, Modeling and Resource Management Research". In: *Proceedings of the 26th IEEE International Symposium on the Modelling, Analysis, and Simulation of Computer and Telecommunication Systems*. MASCOTS '18. Milwaukee, WI, USA, Sept. 2018.

[57] Donald Ervin Knuth. "Literate programming". In: *The computer journal* 27.2 (1984), pp. 97–111.

[58] *Kompose - Convert your Docker Compose file to Kubernetes or OpenShift*. https://kompose.io/. Accessed: 2023-09-30.

[59] Dirk Krafzig, Karl Banke, and Dirk Slama. *Enterprise SOA: service-oriented architecture best practices*. Prentice Hall Professional, 2005.

[60] Rodrigo Laigner, Yongluan Zhou, Marcos Antonio Vaz Salles, Yijian Liu, and Marcos Kalinowski. "Data management in microservices: state of the practice, challenges, and research directions". In: *Proc. VLDB Endow.* 14.13 (Sept. 2021), pp. 3348–3361. ISSN: 2150-8097. DOI: 10.14778/3484224.3484232. URL: https://doi.org/10.14778/3484224.3484232.

[61] Shanshan Li, He Zhang, Zijia Jia, Chenxing Zhong, Cheng Zhang, Zhihao Shan, Jinfeng Shen, and Muhammad Ali Babar. "Understanding and addressing quality attributes of microservices architecture: A Systematic literature review". In: *Information and Software Technology* 131 (2021), p. 106449. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2020.106449. URL: https://www.sciencedirect.com/science/article/pii/S0950584920301993.

[62] Yongkang Li, Yanying Lin, Yang Wang, Kejiang Ye, and Chengzhong Xu. "Serverless Computing: State-of-the-Art, Challenges and Opportunities". In: *IEEE Transactions on Services Computing* 16.2 (2023), pp. 1522–1539. DOI: 10.1109/TSC.2022.3166553.

[63] Fowler Martin. *Bounded Context*. https://martinfowler.com/bliki/BoundedContext.html. Accessed: 2023-09-30. 2014.

[64] Fowler Martin. *Microservices, a definition of this new architectural term*. http://web.archive.org/web/20080207010024/http://www.808multimedia.com/winnt/kernel.htm. Accessed: 2023-09-30. 2014.

[65] Manuel Mazzara, Nicola Dragoni, Antonio Bucchiarone, Alberto Giaretta, Stephan T. Larsen, and Schahram Dustdar. "Microservices: Migration of a Mission Critical System". In: *IEEE Transactions on Services Computing* 14.5 (2021), pp. 1464–1477. DOI: 10.1109/TSC.2018.2889087.

[66] N. Medvidovic and R.N. Taylor. "A classification and comparison framework for software architecture description languages". In: *IEEE Transactions on Software Engineering* 26.1 (2000), pp. 70–93. DOI: 10.1109/32.825767.

[67] Philippe Merle. *KubeDiagrams: Generate Kubernetes architecture diagrams*. https://github.com/philippemerle/KubeDiagrams. GitHub repository, retrieved on 2025-08-22. 2025.

[68] Antonio Messina, Riccardo Rizzo, Pietro Storniolo, Mario Tripiciano, and Alfonso Urso. "The Database-is-the-Service Pattern for Microservice Architectures". In: *Information Technology in Bio- and Medical Informatics*. Ed. by M. Elena Renda, Miroslav Bursa, Andreas Holzinger, and Sami Khuri. Cham: Springer International Publishing, 2016, pp. 223–233. ISBN: 978-3-319-43949-5.

[69] Microservice-API-Patterns. *LakesideMutual*. https://github.com/Microservice-API-Patterns/LakesideMutual. GitHub repository, retrieved on 2025-08-22. 2025.

[70] Hugo Monfleur and Philippe Merle. "Towards Concern-Oriented Microservice Architecture". In: *Proceedings of 2023 International Conference on Microservices*. Universita di Pisa and Microservices Community. Pise, Italy, Oct. 2023. URL: https://inria.hal.science/hal-04201189.

[71] Giuseppe Muntoni and Jacopo Soldani. *microMiner: A tool for automatically generate a microTOSCA specification of a microservice-based architecture*. https://github.com/di-unipi-socc/microMiner. Accessed: 2025-08-20. 2025.

[72] Giuseppe Muntoni, Jacopo Soldani, and Antonio Brogi. "Mining the Architecture of Microservice-Based Applications from their Kubernetes Deployment". In: *Advances in Service-Oriented and Cloud Computing*. Ed. by Christian Zirpins, Iraklis Paraskakis, Vasilios Andrikopoulos, Nane Kratzke, Claus Pahl, Nabil El Ioini, Andreas S. Andreou, George Feuerlicht, Winfried Lamersdorf, Guadalupe Ortiz, Willem-Jan Van den Heuvel, Jacopo Soldani, Massimo Villari, Giuliano Casale, and Pierluigi Plebani. Cham: Springer International Publishing, 2021, pp. 103–115. ISBN: 978-3-030-71906-7.

[73] Evangelos Ntentos, Uwe Zdun, Konstantinos Plakidas, Sebastian Meixner, and Sebastian Geiger. "Assessing Architecture Conformance to Coupling-Related Patterns and Practices in Microservices". In: *Software Architecture*. Ed. by Anton Jansen, Ivano Malavolta, Henry Muccini, Ipek Ozkaya, and Olaf Zimmermann. Cham: Springer International Publishing, 2020, pp. 3–20. ISBN: 978-3-030-58923-3.

[74] *OASIS Topology and Orchestration Specification for Cloud Applications (TOSCA) Technical Committee*. https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=tosca. Accessed: 20 August 2025. OASIS Open, 2025.

[75] *Overview | Docker Docs*. https://docs.docker.com/compose/compose-file/.

[76] Claus Pahl. "Architectural patterns for microservices: a systematic mapping study". In: *CLOSER 2018: Proceedings of the 8th International Conference on Cloud Computing and Services Science; Funchal, Madeira, Portugal, 19-21 March 2018*. SciTePress. 2018.

[77] Sebastiano Panichella, Mohammad Imranur Rahman, and Davide Taibi. "Structural coupling for microservices". In: *arXiv preprint arXiv:2103.04674* (2021).

[78] PiggyMetrics Contributors. *PiggyMetrics: Microservice Architecture Example*. https://github.com/sqshq/piggymetrics. Accessed: 2025-08-05. 2020.

[79] Pkl Authors. *Pkl: A Programming Language for Configuration*. https://pkl-lang.org/. Accessed on August 3, 2025. 2025.

[80] Florian Rademacher. "A Language Ecosystem for Modeling Microservice Architecture". PhD thesis. Kassel, Universität Kassel, Fachbereich Elektrotechnik / Informatik, 2022. DOI: doi:10.17170/kobra-202209306919.

[81] Mohammad Imranur Rahman, Sebastiano Panichella, and Davide Taibi. "A curated dataset of microservices-based systems". In: *Joint of the Summer School on Software Maintenance and Evolution*. CEUR-WS. 2019.

[82] Verified Market Reports. *Microservices Architecture Market Size and Forecast 2024–2033*. Accessed July 2025. 2025. URL: https://www.verifiedmarketreports.com/product/microservice-architecture-market/.

[83] ResearchAndMarkets. *Microservices Architecture Global Market Report 2024*. Accessed July 2025. 2024. URL: https://www.researchandmarkets.com/reports/5782748/microservices-architecture-global-market-report.

[84] Mark Richards. *Microservices vs. service-oriented architecture*. O'Reilly Media Sebastopol, 2015.

[85] Chris Richardson. *A pattern language for microservices*. https://microservices.io/patterns/index.html. Accessed: 2025-07-12. 2025.

[86] Chris Richardson. *FTGO Application: Microservices Example Based on Microservices Patterns*. https://github.com/microservices-patterns/ftgo-application. Accessed: 2025-08-05. 2025.

[87] Chris Richardson. *Microservices patterns: with examples in Java*. Simon and Schuster, 2018.

[88] Jacopo Soldani et al. *microFreshener: A web-based prototype to identify architectural smells and apply refactorings in microservice-based architectures*. https://github.com/di-unipi-socc/microFreshener. Accessed: 2025-08-20. 2025.

[89] Jacopo Soldani, Giuseppe Muntoni, Davide Neri, and Antonio Brogi. "The micro-TOSCA toolchain: Mining, analyzing, and refactoring microservice-based architectures". In: *Software: Practice and Experience* 51.7 (2021), pp. 1591–1621. DOI: https://doi.org/10.1002/spe.2974. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.2974. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2974.

[90] Jacopo Soldani, Damian Andrew Tamburri, and Willem-Jan Van Den Heuvel. "The pains and gains of microservices: A Systematic grey literature review". In: *Journal of Systems and Software* 146 (2018), pp. 215–232. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2018.09.082. URL: https://www.sciencedirect.com/science/article/pii/S0164121218302139.

[91] Spring Team Contributors. *Spring PetClinic: Microservices Version*. https://github.com/spring-petclinic/spring-petclinic-microservices. Accessed: 2025-08-05. 2025.

[92] Alen Suljkanović, Branko Milosavljević, Vladimir Inđić, and Igor Dejanović. "Developing Microservice-Based Applications Using the Silvera Domain-Specific Language". In: *Applied Sciences* 12.13 (2022). ISSN: 2076-3417. DOI: 10.3390/app12136679. URL: https://www.mdpi.com/2076-3417/12/13/6679.

[93] Clemens Szyperski, Dominik Gruntz, and Stephan Murer. *Component software: beyond object-oriented programming*. Pearson Education, 2002.

[94] Davide Taibi. *Microservices_Project_List: A curated list of projects that migrated to or were implemented with a microservices architecture*. https://github.com/davidetaibi/Microservices_Project_List. GitHub repository, retrieved on 2025-08-22. 2025.

[95] Peri Tarr, Harold Ossher, William Harrison, and Stanley M. Sutton. "N Degrees of Separation: Multi-Dimensional Separation of Concerns". In: *Proceedings of the 21st International Conference on Software Engineering*. ICSE '99. Los Angeles, California, USA: Association for Computing Machinery, 1999, pp. 107–119. ISBN: 1581130740. DOI: 10.1145/302405.302457. URL: https://doi.org/10.1145/302405.302457.

[96] Perri Tarr and Harold Ossher. *Concern Spaces*. https://web.archive.org/web/20080122043747/http://www.research.ibm.com/hyperspace/ConcernSpaces.htm. Accessed: 2023-09-30. 2008.

[97] Branko Terzic, Vladimir Dimitrieski, Slavica Kordic, Gordana Milosavljevic, and Ivan Lukovic. "Microbuilder: A model-driven tool for the specification of rest microservice architectures". In: *International Conference on Information Society and Technology*. 2017, pp. 179–184.

[98] *The C4 model for visualizing software architecture*. https://c4model.com/.

[99] Hans Van Vliet, Hans Van Vliet, and JC Van Vliet. *Software engineering: principles and practice*. Vol. 13. John Wiley & Sons Hoboken, NJ, 2008.

[100] Various Authors. *Articles citing the TeaStore microservice reference application*. https://scholar.google.com/scholar?cites=7828615202992379775. Accessed on August 3, 2025. 2025.

[101] Steve Vestal. *A cursory overview and comparison of four architecture description languages*. Tech. rep. Citeseer, 1993.

[102] Mario Villamizar, Oscar Garcés, Harold Castro, Mauricio Verano, Lorena Salamanca, Rubby Casallas, and Santiago Gil. "Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud". In: *2015 10th Computing Colombian Conference (10CCC)*. 2015, pp. 583–590. DOI: 10.1109/ColumbianCC.2015.7333476.

[103] Edwin van Wijk. *pitstop*. https://github.com/EdwinVW/pitstop. GitHub repository, retrieved on 2025-08-22. 2025.

[104] Ben Wilcock. *cqrs-microservice-sampler*. https://github.com/benwilcock/cqrs-microservice-sampler. GitHub repository, retrieved on 2025-08-22. 2025.

[105] Philip Wizenty, Francisco Ponce, Florian Rademacher, Jacopo Soldani, Hernán Astudillo, Antonio Brogi, and Sabine Sachweh. "Towards resolving security smells in microservices, model-driven". In: *18th International Conference on Software Technologies (ICSOFT)*. 2023.

[106] Eberhard Wolff. *microservice-consul: Microservices Example Using Consul for Service Discovery*. https://github.com/ewolff/microservice-consul. Accessed: 2025-08-05. 2020.

[107] Eberhard Wolff. *Microservices: flexible software architecture*. Addison-Wesley Professional, 2016.

[108] *xOpera (opera): A lightweight TOSCA orchestrator*. https://github.com/xlab-si/xopera-opera. Accessed: 2025-08-21. 2025.

Appendix A

Microservice Architecture Design Patterns

A.1 Diversity of Design Patterns

Adopting the microservice architecture style has impacts of different natures going from team organization to infrastructure management through code design and security concerns. Because of that a number of design patterns of different natures have emerged. Table A.2 presents an overview of well known design patterns used in microservices applications categorized by they overarching concern [87] and is followed by a short description for each of them.

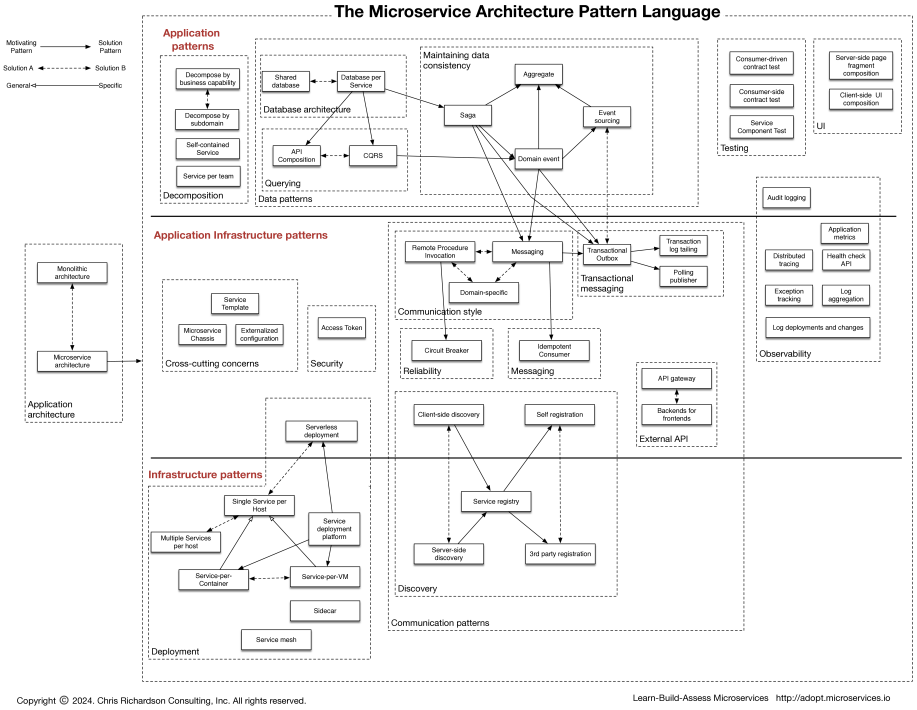


Figure A.1: Microservice Architecture Patterns Topology [85]

Category	Concern	Patterns
Application Patterns	Decomposition	Decompose by Business Capability, Decompose by Subdomain, Self-contained Service, Service per Team
	Data Patterns	Database Architecture: Shared DB vs DB per Service; Querying: API Composition vs CQRS; Maintaining Data Consistency: Saga, Aggregate, Event Source, Domain Event
	Testing	Consumer-Driven Contract Test, Consumer-Side Contract Test, Service Component Test
	UI	Server-Side Fragment Composition, Client-Side Fragment Composition
	Observability	Audit Logging
Application Infrastructure Patterns	Cross-cutting Concerns	Service Template, Microservice Chassis, Externalized Configuration
	Security	Access Token
	Deployment	Serverless Deployment
	Communication Patterns	Transactional Messaging: Transactional Outbox, Log Tailing, Polling Publisher; Communication Style: Messaging vs RPC vs Domain-Specific; Reliability: Circuit Breaker; External API: API Gateway vs Backends for Frontends
	Observability	Application Metrics, Distributed Tracing, Health Check API, Exception Tracking, Log Aggregation, Log Deployment and Changes
Infrastructure Patterns	Deployment	Single Service per Host vs Multiple Services per Host, Service Deployment Platform, Service-per-Container vs Service-per-VM, Sidecar, Service Mesh
	Communication Patterns: Discovery	Service Registry, Server-Side Discovery, 3rd Party Registration

Table A.2: Microservices design patterns

A.2 Application patterns

Decomposition Ways of decomposing an application into microservices.

- **Decompose by Business Capability:** Model the microservice application on the business activity of the company without spreading capabilities among multiple microservices but instead allocating one or several capabilities to strictly one microservice.
- **Decompose by subdomain:** Model the microservice application on the different subdomains that are key to it functioning. Some subdomains may not be mappable to business capability but necessary to perform business actions (e.g. specialized mathematical operations, data storage...). In such a decomposition such sets of related functionalities are grouped and implemented as a service.
- **Self-contained service:** Service does no wait on neighbor's answers for continuing its execution, queries are logged into a database so that even though a microservice can be unavailable it will be able to recover missed queries from it.
- **Service per team:** Each microservice is assigned to a unique team of a relatively small size.

A.3 Data Patterns

- **Database Architecture:**
  - **Shared Database:** different microservices sharing databases. Usual ACID transactions are used making the system simplier to design, develop and operate. On the other hand independance of microservices is reduced limiting the gains microservice architecture brings.
  - **Database per Service:** each database is owned by a single microservice. The pattern keeps microservices loosely coupled, preserving team independance. On the other hand queries distributed among multiple databases are made considerably more difficult.
- **Querying:**
  - **API Composition:** In a context of "database per service" pattern. A specific microservice (e.g. API gateway) queries multiple microservices with dedicated databases to form a composed response.
  - **Command Query Responsability Segregation:** In a context of "database per service" pattern. Modifying actions and querying are managed by different microservices. The microservice answering queries has a database that mirrors

the modifiable database. The query database can unify multiple microservices dedicated databases, keeping independence intact while alleviating the network load of distributed queries.

- **Maintaining Data Consistency:**
  - **Saga:** In a context of "database per service" pattern. For actions that implies multiple microservices, decompose the action in a sequence of smaller actions each local to one microservice. Apply the modifications at each step as long as the local action is successful. The failure of a local action stops the sequence and starts another sequence of compensating actions that undoes the preceding modifications.
  - **Event Source:** Instead of having objects being the composition of multiple current data states, they are the result of series of events emitted by microservices to an event store. It solves the problem of having events published in the wrong order by saving every past event. A problematic effect of using event sourcing is the request complexity and request load of constructing objects from all past events. Can be used with the SAGA pattern and usually needs the CQRS pattern to alleviate load on the event store.

A.4 Testing

- **Service Integration Contract Test:** Tests are written by the teams using the microservices, not the team developing it.
- **Service Component Test:** Having test doubles (analogous to mock class) for every microservice invoked as a result of test a particular microservice.

UI

- **Server-Side fragment composition:** Composition of the displayed webpage is done Server-Side.
- **Client-Side Fragment Composition:** Composition of the displayed webpage is done Client-Side.

Observability Audit Logging: Record user activity in databases.

A.5 Application Infrastructure Patterns

Cross-cutting Concerns

- **Service Template:** Having a simple runnable service whose code is the basis to expand on in order to implement needed functionalities. It serves as a way to speed up production while having with a starting point implementing best practices.
- **Microservice Chassis:** A framework for microservice development that is an assembly of technologies implementing of the shelf functionalities that are relevant to microservice architecture. This framework makes updating libraries easy and standardized.
- **Externalized Configuration:** Having all configurations elements dissociated from application by having values stored in configuration files. The benefits are that changing the environment does not imply code modification and separation of business and configurations concerns.

A.6 Security

- **Access Token:** In an api-gateway pattern context. Having the gateway manage the initial authentication then passing a token to microservices so that they can verify through a centralized credential store if the request is legitimate (authenticity check and access control).

A.7 Deployment

- **Serverless Deployment:** Using a provider that furnishes functions entirely hiding the underlying infrastructure. Can be used as an alternative to microservice architecture, as a particular implementation or in combination with it. It provides some easiness in usage, scaling performance and possibly reduces costs. On the downside it reduces technology freedom, limits greatly infrastructure configuration and can be costly in terms of latency.

A.8 Communication Patterns

- **Transactional Messaging**
  - **Transactional Outbox (or application events):** To publish an event indicating a change in a database, using an auxiliary co-located database (name outbox) that store the changes. The modification of the database and the outbox are made atomically so published events and database state are assured to be eventually consistent. Two ways of publishing transactions are possible :
    - **Polling Publisher:** A service is regularly polling the outbox to publish new transactions to a message broker.

    * **Transactional Log tailing:** Another co-located process mines the outbox transaction logs an then publishes changes to the message broker.

- **Communication Style :**
  - **Messaging:** Asynchronous communication done through sending messages - documents, commands or events - over channels i.e. abstraction of the infrastructure representing ways to communicate. Although messages can be sent directly from one microservice to another, a common form of messaging is to use a message broker. The message broker is a service that acts as an intermediary, storing messages until recipients consumes it. That way location and availability are not important to the sender.
  - **Remote Procedure Invocation:** Synchronous, point to point, communication used to invoke functionalities.
- **Reliability:** Circuit Breaker: In a synchronous communication context. If a number of requests have failed, a timeout is applied during which all incoming requests are rejected.
- **External API:**
  - **API Gateway:** Having a microservice being the single entrypoint for the entire applications.
  - **Backends for Frontends:** Having a different microservice as entrypoint for each kind of clients all build on top a common API layer to interact with the backend.

A.9 Observability

- **Application Metrics:** Services gather metrics during execution. The metrics are then pulled by a server that collects all microservices metrics or pushed by each microservice to the metrics server.
- **Distributed Tracing:** When treating an external request, assign an id to it that is transmitted to every microservice involved in the request. When producing logs, the microservices save the request id so that the trace of the request throughout the system can be identified.
- **Log Aggregation:** Having a service centralizing the logs of every other service in the application.
- **Exception Tracking:** Having a centralized exception service that receives exceptions from all services as they occur.

- **Health Check API:** Having services expose a health endpoints indicating different values about its status. A centralizing service invokes all health endpoint to have a global view of the status of the application.
- **Log Deployment and Changes:** Log every deployment and changes in the application so that errors can be put in relation to them.

A.10 Infrastructure Patterns

- **Deployment**
  - **Single Service per Host:** The service is deployed on a dedicated host (physical or virtual) ensuring isolation and resource availability at the cost of resource efficiency.
  - **Multiple Services per Host:** The host (physical or virtual) runs multiple services allowing better resource efficiency at the cost of isolation and resource availability.
  - **Service Deployment Platform:** Using a deployment solution providing in built functionalities e.g. load balancing, scaling, service discovery...
  - **Sidecar:** Having another process running alongside a service that handles a particular, often technical, aspect.
  - **Service Mesh:** Having a sidecar proxy for all microservices instances that manages a range of tasks relative to networking. A centralized control plane component manages the proxies allowing to control network at application level.
- **Communication Patterns:**
  - **Service Registry:** A database of microservices running instances that is queried to allow communications. The service registry is often responsible for performing health checks so that it provides only locations of available microservices instances. Microservices need to be registered in the service discovery and can do so in two ways:
    * **Self Registration:** The microservices are responsible for registering themselves in the service registry at startup.
    * **Third Party Registration:** A dedicated process or service is responsible for registering the service at startup. While making the architecture more complex, it lightens the code of the application.

    Different patterns can be used relatively to how to query the service registry for service discovery:

- **Service-Side Discovery:** The service directly queries the service registry to discover the location of a target service. Services host a local registry to limit the network load of querying the service registry.
- **Server-Side Discovery:** The service transmit its request to a router that is responsible for locating and transmitting requests to the target services.

Those patterns give an idea of what could be expressed in a microservice ADL as they are architectural choices. Being able to identify which ones are being used and which microservices pertain to them would be a way not only to give an understanding of the application logic but also to state and ensure constrains that span over multiple microservices.

Appendix B

Toolbox

This section describes the different functionalities of the toolbox to be used on `.comsa` files. With compiling, analysis and graphical representation, it provides complete support for the COMSA language.
Installation and usage of the toolbox is described in the COMSA Github repository [1] [42]. It is proposed as a Docker image that can either be pulled from DockerHub or built from the GitHub repository or as an executable Python script named `comsatools`. The commands presented in the following sections use the `comsatools` executable.

B.1 Compilers

The central components of the toolbox are the compilers. Docker Compose and Kubernetes have been selected as target languages because of their popularity, but more importantly because of their declarative nature and executability which, through compilers makes COMSA an executable language. Since all expressible properties in both target languages are expressible in the COMSA language it makes COMSA a possible alternative for them.

B.1.1 `comsa2compose-yaml`

Usage

Compile the application expressed in COMSA toward the Compose language with YAML concrete syntax. Because of that it is directly executable using Docker Compose.

Example Command:

```
comsatools comsa2compose-yaml examples/teastore.comsa
```

Output:

[1] https://github.com/hmonfleur/comsa

```
version: '3'
services:
  persistence:
    depends_on:
      db:
-------------------------------------------------------------
    - '3306'
    image: descartesresearch/teastore-db
    ports:
    - 3306:3306
networks: {}
```

Implementation

Listing B.1 shows how to structure the Pkl schema to produce a Docker Compose file from a `.comsa` file.

```
import "$(pkl-uri $1)" as template
tmp_concerns= (template.computeConcerns(template.concerns))
output {
  renderer = new YamlRenderer {}
  value = new {
    version = template.version
    services = template.computeServices(template.services, tmp_concerns)
    networks = template.computeNetworks(tmp_concerns)
    volumes = template.computeVolumes(template.volumes, tmp_concerns)
    secrets = template.secrets
    configs = template.configs
  }
}
```

Listing B.1: Schema for to convert `.comsa` files to Docker Compose files.

The process consists in adding an extension to the original file that defines an `output` section defining the structure of the output of the pkl evaluation. The format of the file is deteminated by instantiating a Pkl built-in renderer class. Here the class being `YamlRenderer` the result will be a Yaml file. The semantical content of the produced file is contained in the `value` attribute where the appropriate root attributes of a Docker Compose file are indicated. The most important value is the one assigned to the `services` attribute that uses the function `computeServices` to generate compose services from the `Concern` objects declared in the `.comsa` file.

```
function computeServicesModified(
    service_mapping: Mapping<String, Service>?,
    concern_mapping: Mapping<String, Concern>?)
    : Mapping<String, Service>
```

Listing B.2: `compute_services` function in `comsa` module

Listing B.2 shows the signature of the `computeServices` function, indicating that it takes a `Mapping` with `Concerns` as values and returns a `Mapping` with `Services` i.e.

Docker Compose services. The usage of the function in Listing B.1 consists in merging all the partial definitions of services declared as parts of `Concern` objects in the `.comsa` file. That function is applied to the result of the `computeConcerns` function whose result has been stored in `tmp_concerns` that simply adds to the other declared concerns the ones declared through `Multiconcerns` objects.

B.1.2 `comsa2compose`

The `comsa2compose` command also compiles `.comsa` files to the Compose language but keeps the Pkl syntax instead of adopting the Yaml syntax. Its logic is identical to `comsa2compose-yaml` but does not use a `YamlRenderer`.

Example Command:

```
comsatools comsa2compose examples/teastore.comsa
```

Output:

```
version = "3"
services {
  ["persistence"] {
    deploy = null
    build = null
------------------------------------------------------------
        `x-sends_to` = null
      }
    }
  }
}
```

B.1.3 `comsa2yaml`

The `comsa2yaml` command compiles the `.comsa` file to the COMSA language but using the Yaml syntax. Its logic is identical to `comsa2compose-yaml` but instead of producing a Compose description it produces a COMSA description simply not using the `computeServices` function and having a `concerns` attribute.

Example Command:

```
comsatools comsa2yaml examples/teastore.comsa
```

Output:

```
version: '3'
concerns:
  DescartesresearchTeastore Images:
    services:
      persistence:
-------------------------------------------------------------
    main_service: {}
    main_service_aliases:
    - {}
    service_aliases:
    - {}
```

B.1.4 comsa2k8s

Compile the application expressed in COMSA-Pkl to a Kubernetes manifest with YAML concrete syntax. Because of that it is directly executable using Kuberbetes.

Usage

Example Command:

```
comsatools comsa2k8s examples/teastore.comsa
```

Output:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: teastore
-------------------------------------------------------------
          service:
            port:
              number: 3306
            name: db
        pathType: Prefix
```

Implementation

Listing B.3 shows how to structure the Pkl schema to produce a Kubernetes manifest from a `.comsa` file.

```
amends "modulepath:/compose2k8s.pkl"
import "$(pkl-uri $1)" as template
resources = module.compose2k8s("$(basename -s .comsa $1)", template.
    computeServicesModified(template.services, template.computeConcerns(
    template.concerns)))
```

Listing B.3: Schema to convert `.comsa` files to Kubernetes manifests.

A first step in the conversion is to compute services using the `computeServices` on the application `Concern` objects. The result is then converted to a `Listing` of Kubernetes ressources using the `compose2k8s` function defined in the `compose2k8s.pkl` module and whose signature is shown in Listing B.4.

```
function compose2k8s(
    app_name: String,
    services: Mapping<String, compose.Service>)
    : Listing<K8sResource>
```

Listing B.4: Signature of the `compose2k8s` function.

The function makes use of the Pkl built-in definition of the Kubernetes ressources.

B.1.5 `convert2pkl`

Convert a Compose file from a Yaml syntax to a Pkl syntax. The executable is a Python script that requires the PyYaml module.

Example Command:

```
comsatools convert2pkl examples/teastore.yml
```

Output:

```
amends "modulepath:/compose.pkl"
version = "3"
services {
  ["registry"] {
    image = "descartesresearch/teastore-registry"
----------------------------------------------------------------
    ports {
      "8080:8080"
    }
  }
}
```

B.1.6 `compose2yaml`

Compile the Compose application from a Pkl to a YAML concrete syntax. Opposite process to `convert2pkl` using Pkl `YamlRenderer` built-in object.

Example Command:

```
comsatools compose2yaml examples/teastore.pkl
```

Output:

```
version: '3'
services:
  registry:
    expose:
    - '8080'
------------------------------------------------------------
    expose:
    - '8080'
    image: descartesresearch/teastore-webui
    ports:
    - 8080:8080
```

B.1.7 `compose2k8s`

Compile the application expressed in Compose-Pkl to a Kubernetes manifest with YAML concrete syntax. Corresponds to the last step of comsa2k8s.

Example Command:

```
comsatools compose2k8s examples/teastore.pkl
```

Output:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: teastore
------------------------------------------------------------
          service:
            port:
              number: 8080
            name: webui
        pathType: Prefix
```

B.2 Analysis

B.2.1 `comsa-analysis`

Static analysis of a COMSA-Pkl file, provides architectural informations.

Example Command:

```
comatools comsa-analysis examples/teastore.comsa
```

Output:

```
File: examples/teastore.comsa
Services are 'persistence', 'auth', 'image', 'recommender', 'webui', 'registry
    ', 'db'.
Technical services are 'webui', 'db', 'registry'.
Business services are 'persistence', 'auth', 'image', 'recommender'.

------------------------------------------------------------
[Info] No application metrics pattern.
[Info] No distributed tracing pattern.
[Info] No shared database pattern.
[Warning] 'auth', 'image', 'recommender' have not its own database!
[Info] No CQRS pattern.
```

B.2.2 `comsa-check SOURCE`

Static analysis of a COMSA-Pkl file, provides architectural informations, warnings and errors.

Example Command:

```
comsatools comsa-check examples/teastore.comsa
```

Output:

```
File: examples/teastore.comsa
[Info] services.persistence: neither an image nor a build context specified
[Info] services.persistence: depends on undefined service
[Info] services.persistence: no healthcheck configured
[Info] services.persistence: refers to undefined network
------------------------------------------------------------
[Info] services.db: refers to undefined secret
[Info] services.db: ‘scale‘ is deprecated. Use the ‘deploy.replicas‘ element
[Info] services.db: can't use both 'scale' (deprecated) and 'deploy.replicas'
0 errors
0 warnings
```

B.2.3 `comsa-metrics SOURCE`

Static analysis of a COMSA-Pkl file, provides metrics relative to concerns used or unused in the application.

Example Command:

```
comsatools comsa-metrics examples/teastore.comsa
```

Output:

```
file = "examples/teastore.comsa"
statistics {
  concerns = 6
  property_centric_concerns {
    all = 2
------------------------------------------------------------
    }
    other_patterns = 0
  }
  other_concerns = 0
}
```

B.2.4 comsa-scattering

Static analysis of a COMSA-Pkl file, provides metrics relative to the scattering of individual Concerns in the application. Can be used with the `-v` flag for more detailed information.

Example Command:

```
comsatools comsa-scattering examples/teastore.comsa
```

Output:

```
scattering {
  ["DescartesresearchTeastore Images"] = 100
  ["Registry"] = 86
  ["WebUI"] = 71
  ["Persistence"] = 43
  ["Database Per Service Pattern"] = 29
  ["Public Ports"] = 29
}
```

B.2.5 comsa-tangling

Static analysis of a COMSA-Pkl file, provides metrics relative to the tangling of individual Services i.e. the degree to which Services are shared among Concerns. Can be used with the `-v` flag for more detailed information.

Example Command:

```
comsatools comsa-tangling examples/teastore.comsa
```

Output:

```
tangling {
  ["persistence"] = 83
  ["webui"] = 67
  ["recommender"] = 67
  ["image"] = 67
  ["db"] = 50
  ["auth"] = 50
  ["registry"] = 33
}
```

B.2.6 comsa-type-scattering

Static analysis of a COMSA-Pkl file, provides metrics relative to the scattering of types of Concerns in the application. Can be used with the `-v` flag for more detailed information.

Example Command:

```
comsatools comsa-type-scattering examples/teastore.comsa
```

Output:

```
scattering_by_type {
  ["ImageConcern"] = 100
  ["ServiceRegistryPattern"] = 86
  ["ApiGatewayPattern"] = 71
  ["SharedService"] = 43
  ["DatabasePerServicePattern"] = 29
  ["PublicPortsConcern"] = 29
}
```

B.2.7 compose-metrics

Static analysis of a Compose file in YAML concrete syntax, provide global metrics relative to properties.

Example Command:

```
comsatools compose-metrics examples/teastore.yml
```

Output:

```
File: examples/teastore.yml
#services = 7
#properties = 28
#depends_on = 0
#links = 0
#x-connects_to = 0
#x-sends_to = 0
#similar properties = 23
#unique properties = 9
```

B.2.8 `compose-check`

Static analysis of a Compose file in Pkl concrete syntax, provides architectural informations, warnings and errors.

Example Command:

```
comsatools compose-check examples/teastore.pkl
```

Output:

```
File: examples/teastore.pkl
[Info] services.registry: neither an image nor a build context specified
[Info] services.registry: depends on undefined service
[Info] services.registry: no healthcheck configured
[Info] services.registry: refers to undefined network
------------------------------------------------------------
[Info] services.webui: refers to undefined secret
[Info] services.webui: ‘scale‘ is deprecated. Use the ‘deploy.replicas‘
    element
[Info] services.webui: can’t use both ’scale’ (deprecated) and ’deploy.
    replicas’
0 errors
0 warnings
```

B.3 Graphical Representation

B.3.1 `topology`

Generate an image of a graph representation of the topology from the architectural patterns used in a COMSA file in `.comsa` format. Since we already provided the Teastore topology graphical representation in Figure 5.1 we present the graphical representation of the Social Network application which is considerably larger.

Example Command:

```
comsatools topology examples/deathstarbench_socialnetwork.comsa
```

Generated image:

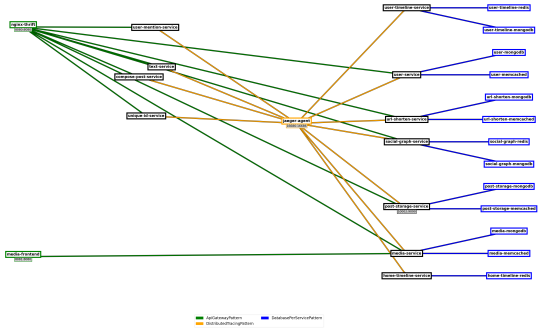


Figure B.1: Representation of the topology of the Social Network application.

B.3.2 `hypergraph`

Generate an image of an hypergraph representation of the concerns in a COMSA file in `.comsa` format.

Example Command:

```
comsatools hypergraph examples/teastore.comsa
```

Generated image:

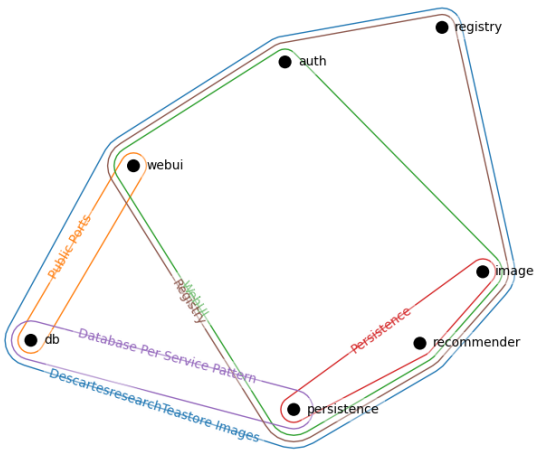


Figure B.2: Hypergraph representation of the Teastore application.

B.3.3 `hypergraph-policies`

Generate an image of an hypergraph representation of the policies in a COMSA file in `.comsa` format from the property centric concerns used in it.

Example Command:

```
comsatools hypergraph-policies example/open5gs.comsa
```

Generated image:

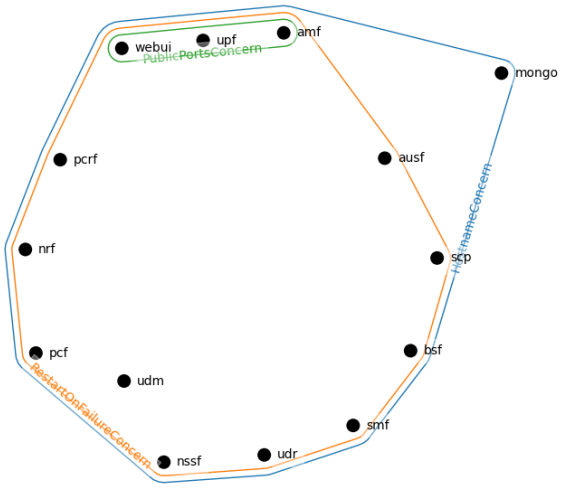


Figure B.3: Hypergraph representation of the policies in the Open5gs application.

B.3.4 `compose2diagram`

Generate an image from a Compose file in YAML format, represent the services, their deployment dependency (using the `depends_on` attribute) and their online connection (using the `x-connect-to` attribute).

Example Command:

```
comsatoosl compose2diagram examples/teastore.yml /tmp
```

Generated image:

Figure B.4: Diagram of the Teastore application from original file.

The generated image is less than satisfying. That's because the original description of the Teastore applications in Compose YAML does not contain any `depends_on` attributes for services.
However if we use the following command :

```
comsa2compose-yaml examples/teastore.comsa 1> /tmp/
    teastore_compose_from_teastore_comsa.yml
```

We generate the Compose YAML file with generation of `depends_on` and `x-connect-to` attributes that are implicit in most `MicroserviceArchitecturePattern` concerns. We can then use the following command to generate the diagram:

```
compose2diagram /tmp/teastore_compose_from_teastore_comsa.yml /tmp
```

Generated image:

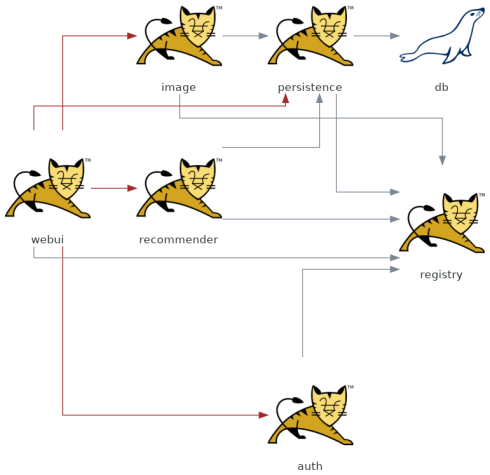


Figure B.5: Diagram of the Teastore application from Compose file generated from COMSA file.
Gray lines represent deployment dependency. Red lines represent online dependency.

B.4 Tests

B.4.1 `eval_all`

Tests most of the previously mentioned on the files in the `dataset` folder and stores the outputs in the `generated` folder (created if not present). Indicates faulty commands. The executable is located in the `tests` folder and can be launched using the following command from the root folder:

```
./tests/eval_all
```

Output:

```
1  Convert Compose yml files to Compose pkl
2  convert2pkl on 31 files:
3  DONE
   ------------------------------------------------------------
28 Convert pkl comsa files to yaml comsa files
29 comsa2yaml on 25 files:
30 DONE
```

B.5 IDE integration

Since COMSA is written with Pkl language, it benefits from the entire Pkl toolchain. Notably Pkl provides an excellent IDE plugin for Intellj IDEA [49] into which the comsa module can be imported to provide syntax highlighting, checking and text completion. Support for other text editors have been developed but is less extended.

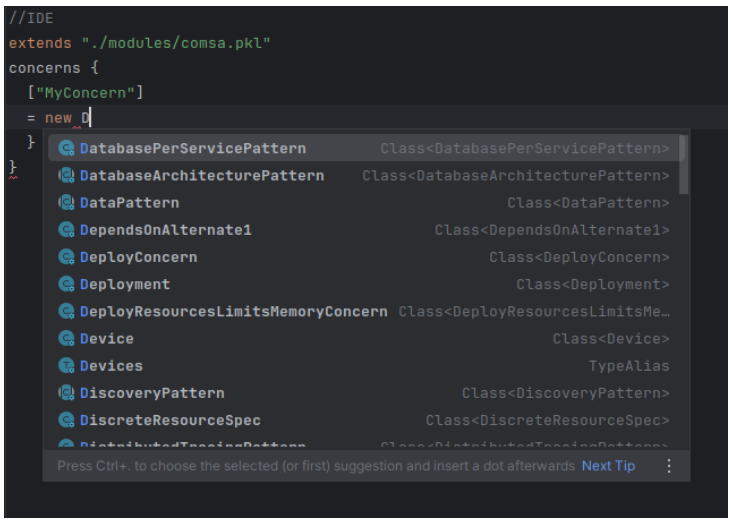
It is first necessary to install the Pkl plugin for Intellij IDEA. To do so refer to the toolbox section of the Pkl Language website[2]. Then you need to locally have a folder on your computer containing the files `comsa.pkl` and `compose.pkl`. And begin your COMSA file with the following lines:

```
1 //IDE
2 extends "/path/to/comsa.pkl"
```

Listing B.5: Import for IDE support of .comsa files.

Note that the `//IDE` (alternatively `// IDE`) is **mandatory**. It instructs `comsatools` to remove this line.

You can now enjoy complete IDE support.



[2] https://pkl-lang.org/intellij/current/installation.html