Towards Recovering the Software Architecture of Microservice-based Systems

Giona Granchelli*, Mario Cardarelli*, Paolo Di Francesco†,
Ivano Malavolta‡, Ludovico Iovino†, Amleto Di Salle*
*University of L'Aquila, L'Aquila, Italy
{giona.granchelli | mario.cardarelli}@student.univaq.it, amleto.disalle@univaq.it
†Gran Sasso Science Institute, L'Aquila, Italy - {paolo.difrancesco | ludovico.iovino}@gssi.it
‡Vrije Universiteit Amsterdam, The Netherlands - i.malavolta@vu.nl

***Abstract*—Today the microservice architectural style is being adopted by many key technological players such as Netflix, Amazon, The Guardian. A microservice architecture is composed of a large set of small services, each running in its own process and communicating with lightweight mechanisms (often via REST APIs). If on one side having a large set of independently developed services helps in terms of developer productivity, scalability, maintainability, on the other side it is very difficult to have a clear understanding of the overall architecture of a microservice-based software system, specially when the deployment and operation of the involved microservices evolves at run-time.**

In this paper we present MicroART, an architecture recovery approach for microservice-based systems. By using Model-Driven Engineering techniques, we leverage a suitably defined domain-specific language for representing the key aspects of the architecture of a microservice-based system and provide a toolchain for automatically extracting architecture models of the system. The only inputs of MicroART are: (i) a GitHub repository containing the source code of the system and (ii) a reference to the container engine managing it. We validated MicroART on a publicly available benchmark system, with promising results.

***Index Terms*—Microservices, Architecture recovery, Model-Driven Engineering.**

I. Introduction

Lewis and Fowler define the microservice architectural (MSA) style as an approach to developing a single system as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API [10]. This style puts emphasis on the design and development of highly maintainable and scalable software where large systems are decomposed into independent services [18]. Services are small in size with respect to Service-Oriented Architecture (SOA) design, independent from each other, and designed using bounded contexts to combine together related functionalities [9]. Alshuqayran et al. [1] state that commonly agreed on benefits of this style are multiple, e.g. increase in agility, developer productivity, resilience, scalability, reliability, maintainability, separation of concerns, and ease of deployment. However, a set of challenges have significant impact on this style, as services discovering over the network, security management, communication optimization, data sharing and performance [1].

Model-Driven Engineering (MDE) promotes models as first-class entities and leverages the abstraction of software development from coding to modeling [5, 21]. The main benefit is the intention to better manage the increasing complexity of modern software while preserving the values of quality attributes of code-centric techniques. A model is a high-level representation of aspects of a system usually employed by software engineers to precisely specify concepts and relationships before the development phase starts. Modeling tools can be sophisticated and they can even generate the skeleton or all of the code without employing explicit programming techniques that can result particularly error-prone.

Reverse engineering is the process of analyzing and comprehending the software and producing a representation of it at a high level of abstraction [3, 4]. In particular, *Model-driven Reverse Engineering* (MDRE) applies reverse engineering techniques in combination with modeling technologies, to overcome the maintenance problem. Reverse engineering methodologies have many applications and purposes, e.g., design recovery, program comprehension of legacy systems aimed to support the evolution of the system, and description reconstruction of poorly documented systems [4, 17]. One of the main tasks in reverse engineering is *design recovery*. It reduces the complexity of software systems by leveraging the software architecture abstraction instead of dealing directly with the source code [19]. In detail this task recreates design abstractions from a combination of code, existing design documentation when available, personal experience, and general knowledge about application domains [2]. When the process of reverse engineering produces an explicit architecture representation is usually described as *reverse software architecting* process [13]. Reverse Engineering techniques have been largely applied in literature for architecture recovery and change dependency analysis [17, 22], but recent studies confirmed that in the *microservice architectural style* area little investigation is being performed [1, 8].

In this paper we present an architecture recovery approach for microservice-based systems to tackle the problem of the complexity of microservice architecture. The proposed approachis implemented as a prototype named MicroART [11]. More specifically, our approach is able to automatically extract the deployment architecture of a microservice-based system starting from (i) a GitHub repository containing its Docker-based source code and (ii) a reference to the Docker container

engine managing it; then, the approach allows software architects to semi-automatically refine the obtained deployment architecture into the final software architecture of the system. Large microservice-based architectures can grow up to hundreds or thousands of services, each one with its own peculiarities and different running environment. Organizing and managing a high number of services can be challenging and being able to recovery the software architecture from source code repositories or other artefacts might result in a significant improvement in the management and comprehension of this type of systems.

A first assessment of the approach and its prototype validation have been performed on the Acme Air system, an open-source benchmark microservice-based system, confirming the applicability and the advantages of the approach.

The rest of the paper is organized as follows. Section II presents the proposed approach and Section III discusses the details of the MicroART DSL. Section IV presents the validation of MicroART with respect to a third-party benchmark system. Section V presents the implementation of our approach, whereas Section VI discusses related work. Section VII closes the paper and discusses future work.

II. Recovering Microservice-based Architectures

The MicroART approach for microservice architecture recovery is composed of two phases, namely *architecture recovery* labelled with (A) and *architecture refinement* labelled with (B), as shown in Figure 1.

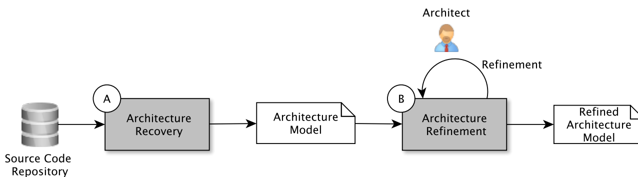


Fig. 1: Approach overview

The *architecture recovery* phase deals with all the activities necessary to extract an *architectural model* of the system starting from its source code repository. From the source code repository we expect to find information about: system (e.g., system and service names), deployment (e.g., service descriptors), product management (e.g., teams, developers, releases). This makes the approach specially tailored for microservice-based systems. Among the most important reasons why we have built our approach on the analysis of repositories there are: *(i)* they enable automation of the process, by providing access to the source code but also system's configuration files, *(ii)* they provide access to the history of the evolution of the system, and *(iii)* they are widely adopted in industry. For *architecture model* we mean a model representing the deployment architecture of the system, which is composed by all the elements of the architecture.

The *architecture refinement* phase aims to refine the initial architecture model into one or more *refined architectural models* by means of model refinement incremental steps. The software architect can decide to enhance the generated architectural model in order to recover an architecture more suitable for its needs, e.g., removing unnecessary details, perform model analysis, architectural change impact analysis, overall understanding of the system, and finally decide when the *refined architectural model* is ready to be finalized. In the remaining of the section we detail the two phases with the related steps to obtain the final result.

A. Architecture Recovery

In the *architecture recovery* phase, we recover the system architecture by analyzing the system's source code repository. It is based on the *extract-abstract-present* paradigm [14], depicted in Figure 2. Generally, in the *Extraction* phase, the information is extracted from artefacts as the system's source code, documentation, history, or the architect knowledge. *Abstraction* is about grouping and filtering information to obtain a meaningful and focused set of information. *Presentation* is about organizing the information in a way that is familiar to the targeted readers [13].

Fig. 2: Architecture recovery: extract-abstract-present paradigm

Figure 3 provides a graphical representation of the architecture **Extraction** activities. The extraction phase is divided in two major activities: static analysis and dynamic analysis.

In the *static analysis*, the following information is retrieved by analysing the repository given as input: *(i)* service descriptors, *(ii)* system name, and *(iii)* developers. Service descriptors are simple text files describing properties and configurations of each service in order to efficiently package and run each service and its dependencies. These files contain all the necessary information related to the deployment of each service in the target environment, as the name of both the service and its container, input and output ports of containers, and the build path. Examples of service descriptors are the Docker and Vagrant files, respectively for the Docker[1] container engine and the Vagrant[2] platform. The system name and the information about each developer which has contributed to the repository are collected, specifically name, username, and email.

Once the static analysis is completed, the *dynamic analysis* activity begins. The dynamic analysis aims to extract the following information: *(i)* containers information, and *(ii)* the communication logs. Since this information is not available statically, this operation must be performed at runtime. Two main steps are involved in the dynamic analysis: *(i)* query the runtime environment in order to identify each container IP address and the network interface used by microservices for their

[1]https://www.docker.com
[2]https://www.vagrantup.com

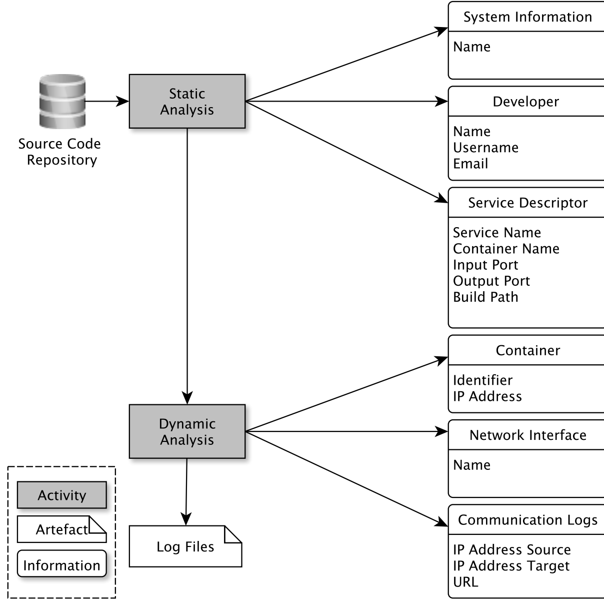


Fig. 3: Detailed activities of the extraction phase

communication, and *(ii)* the creation of the communication logs. Container engines typically provide runtime environment information by running specific commands (e.g., in Docker the command is *inspect*). By means of these commands, the containers information is extracted and the specific network interface used for microservice communications is discovered. By using a monitoring tool (in our implementation TcpDump) this interface can be monitored, and the communication logs can be written to *Log files*.

The **Abstraction** phase is about grouping and filtering the information gathered in the extraction phase in order to rework the collected information in the desired manner. The extracted information is given as input to a *model factory* that creates the *architecture model*, which is the final output of the *architecture recovery* phase as shown in Figure 2. The model generated at this stage conforms to MicroART-DSL, our domain-specific modeling language for microservice-based architectures. The concepts and relationships within the MicroART-DSL language are presented in Section III.

The abstraction phase includes the mapping procedure that associates the gathered data from the previous phase (see Figure 3) to the MicroART-DSL concepts (see Figure 4). As shown in Table I, for each extracted data, a direct mapping to the DSL concept is matched and the result of this association are the models described in section IV.

The first step of the mapping procedure creates an instance of the root metaclass *Product* by setting the extracted system name. For each service descriptor a new *MicroService* class is created, and the corresponding attributes are mapped. For each MicroService class, a new instance of *Team* class will be attached, because a single team is associated to a single microservice. A *Developer* class is created for each developer that has committed to the system repository. On the basis of the commit history and the commit paths, if a match is found with a specific microservice build path, then the *Developer* is assigned to the microservice's team. On the basis of the communication logs, for each microservice communication an *Interface* class is created on both the service provider and the service consumer side. A source interface is introduced in order to keep track of the source service request, instead the target interface will have more detailed information about the resource needed, i.e., endpoint, protocol and method. For each communication, a new translation into a *Link* class will be applied, representing the connection between two interfaces. The creation and the utilization of the *Cluster* metaclass is a classification, not involved in the translation mapping, that can be used to group microservices under specific characteristics, as also discussed in section III, *Cluster* provides a logical division of the system.

The **Presentation** phase is related to render the obtained *architecture model* making the modelled concepts, extracted and mapped in the previous phase available and especially exploitable to the software architect. The *architecture model* extracted can be rendered with a visual editor, or in a text editor developed and distributed with a concrete syntax part of the infrastructure developed in MicroART, like the editors showing the models rendered in Section IV in Figure 6a and 6b.

B. Architecture Refinement

This phase, labelled in Figure 1 with (B) is semi automatic, since it requires the intervention and supervision of the software architect. Initially, a model of the architecture is automatically created, as presented in phase (A). After that, refinements are applied on the model, leading thus to the final microservice architecture representation. We define *refinement* such as the process of modification of the *architecture model* in a new model, in this case, filtered of some of the contained elements that the architect can easily spot. For this phase the architect interaction is needed since it has to select one or more components in the architecture model in order to filter or resolve that specific components from the system representation. The purpose of this phase is produce another *architecture model* which the architect considers more significant for its purposes. This new architecture is referred to as *refined architecture model*, as shown in Figure 1.
The advantages of adopting *refined architecture models* are several, and could be defined considering the architect needs. Indeed, a *refined architecture model* could be used for analysis, or for obtaining different views of the system architecture customized on specific components. The first refinement we have applied to our approach is the *Service Discovery Resolution*, and other refinements can be further defined and integrated in the current platform.

Service Discovery Resolution is the first architectural refinement considered in the approach and its purpose is to resolve the *service discovery* services in order to reveal the dependencies among microservices. Microservice-based

TABLE I: Mapping between the extracted information and the MicroART-DSL

Extracted information			**DSL concept**
Analysis type	**Information**	**Concept**	**Metaclass**
Static analysis	GitHub metadata	System Name	Product (name)
		Developers	Developer (Name, Username, Email)
	Service Descriptor	Service name	Team (Name)
		Container name	Microservice (Name)
		Input ports	Interface (Port)
		Output ports	Interface (Port)
		Build path	Microservice (Build)
Dynamic analysis	Containers	Identifiers	Microservice
		IP address	Microservice (Host)
	Communication Logs	IP source address	Link
		IP target address	Link
		URL	Interface (EndPoint)

architectures adopt service discovery mechanisms in order to keep the microservices dependencies loosely coupled. Since microservices might change their status dynamically (e.g., IP address) for several reasons (e.g., upgrades, autoscaling or failures), the service discovery mechanisms are used to allow services to find each other in the network dynamically. Despite service discovery mechanisms are fundamental to simplify the discover of services at run-time, they mask the real resource dependencies among microservices in the system. The software architect can use the MicroArt graphical editor for identifying and selecting the *service discovery* services in the *architecture model* of the system and obtain a new *refined architecture model* where dependencies are represented.

The *architecture model* is refined into the *refined architecture model* by following three steps: *(i)* removing all the existing *Links* among microservices, *(ii)* removing the service discovery *MicroServices* identified by the software architect, and *(iii)* creating new *Links* by identifying the actual connections between microservices. This last step is feasible because it is now possible to track each request from the service consumer to the targeted service provider.

III. The MicroART Domain-Specific Language

In this section we present the conceived domain-specific language (DSL) for microservice-based systems and discuss its underlying metamodel, as depicted in Figure 4. The metamodel is composed of seven metaclasses where *Product* is the root node of the system being designed. *MicroService* represents the microservices composing the system and its attributes are: host, which is the assigned IP address, and type, which is the type assigned to the microservice. According to the service type classification discussed by Richards [20], we have classified in our DSL the microservices as either functional or infrastructural. The set of possible values of the *ServiceType* enumeration allows to define a service as functional by assigning to the service the value functional, or implicitly define the service as infrastructural by assigning to it one of the remaining possible infrastructural value, which were extracted according to the classification provided in [8]. *Interface* represents a communication endpoint and it is attached to specific microservices, for which it represents either an input or output port. *Link* connects two interfaces together, thus representing the communication among them. *Team* is composed of one or more developers. Each microservice is *owned* by one and only one team. The metaclass *Developer* represents a software developer that participates to the development of the system. *Cluster* is a logical abstraction for grouping together specific microservices.

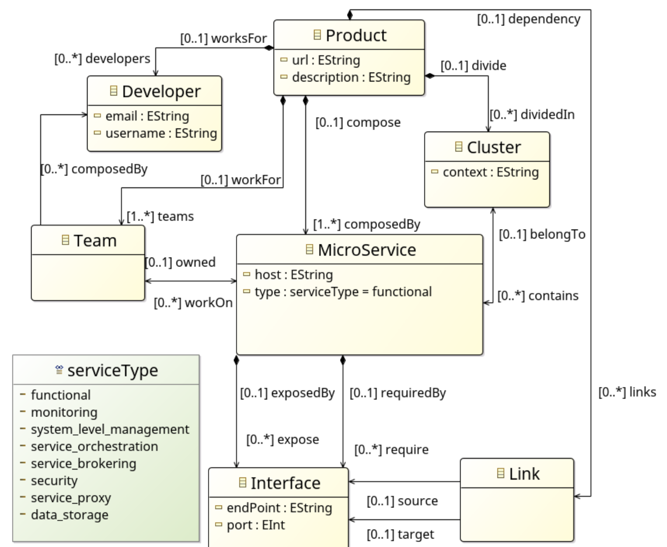


Fig. 4: DSL Metamodel for microservice-based systems

Our DSL has been developed around the microservice needs and characteristics [10], and it is kept minimal in order to support the design and description of multiple microservice-based systems.

Typical aspects of the MSA style represented in the DSL. First, the notion of *products, not projects*[3] where cross-functional teams are responsible for building and operating each product has been realized by tying together the metaclasses Product, Microservice, Team and Developer. Second,

[3]https://www.martinfowler.com/articles/microservices.html#ProductsNotProjects

the Cluster metaclass allows to group together microservice in specific categories, as for example *functional* and *infrastructural* services [15]. Third, the metaclasses Interface and Link allow to define *lightweight* communication protocols, by specifying for their representation only the required basic information.

Every model generated by MicroART is an instance of the MicroART-DSL metamodel; examples of model instances are discussed in Section IV.

IV. Validation

The presented approach and its prototype tool have been applied on a publicly available open-source system called Acme Air[4]. Acme Air is a microservice-based system of a fictitious airline system implemented in NodeJS, and runs on top of the Docker platform[5]. The given input to the MicroART tool is the Acme Air GitHub repository URL and the architecture recovery activity (A) starts. The details of the extraction phase are shown in Figure 5, where the used artefacts are shown simply instantiating the extraction phase presented in Figure 3.

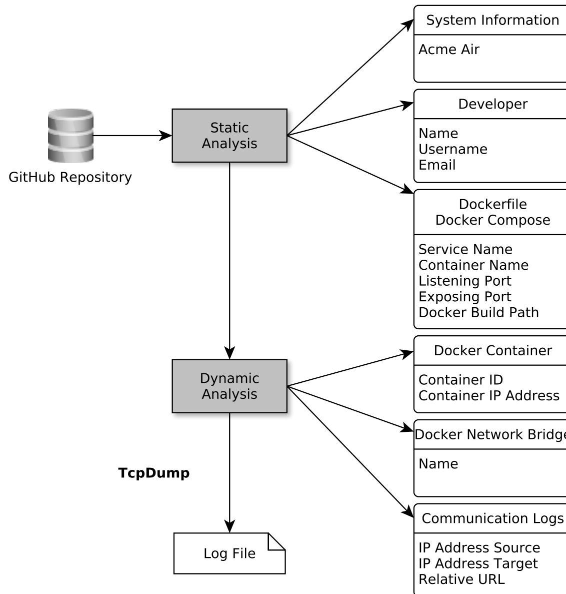


Fig. 5: Acme Air extraction phase

The static analysis extracts the service descriptors, specifically represented by the DockerFiles and the Docker-compose file. The system name (i.e., Acme Air), and the developers information are extracted. The dynamic analysis starts by running the *inspect* command at runtime, from which the container information and the network interface (i.e., Docker Network Bridge) are obtained. Using this information, the running TcpDump monitoring tool detects the communication among services and stores them into log files, while the system usage is simulated.

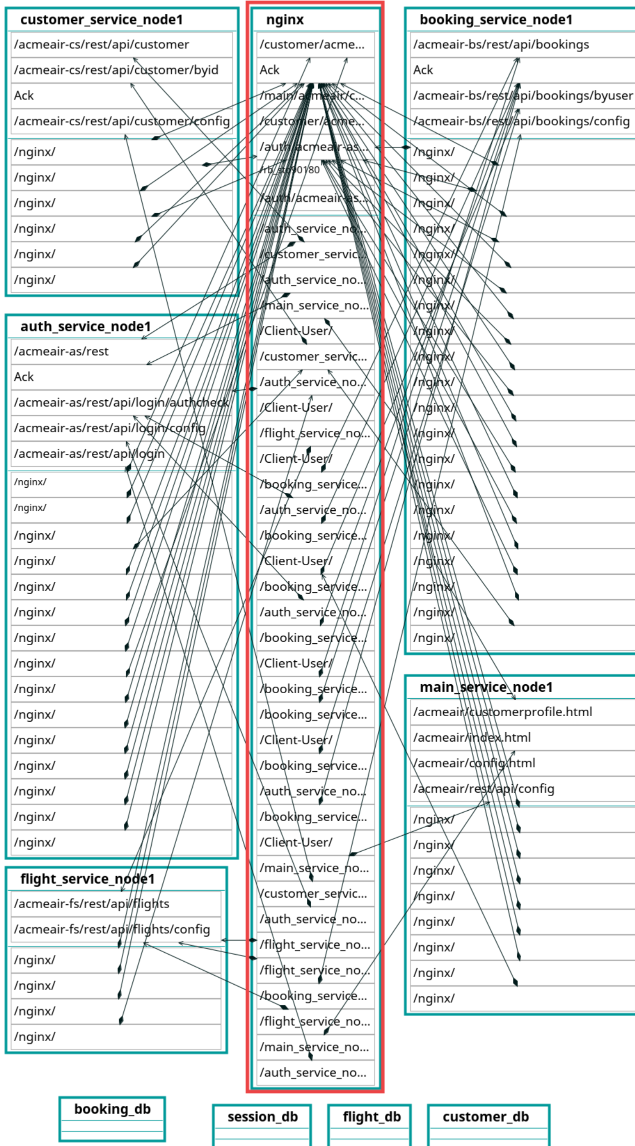
[4]https://github.com/acmeair/acmeair
[5]https://www.docker.com

The second phase of the architecture recovery (i.e., the abstraction phase) begins, the collected information is mapped to the DSL metaclasses and the architecture model is instantiated. The resulting model is visualized with the help of the developed graphical editor and a screenshot is shown in Figure 6a.

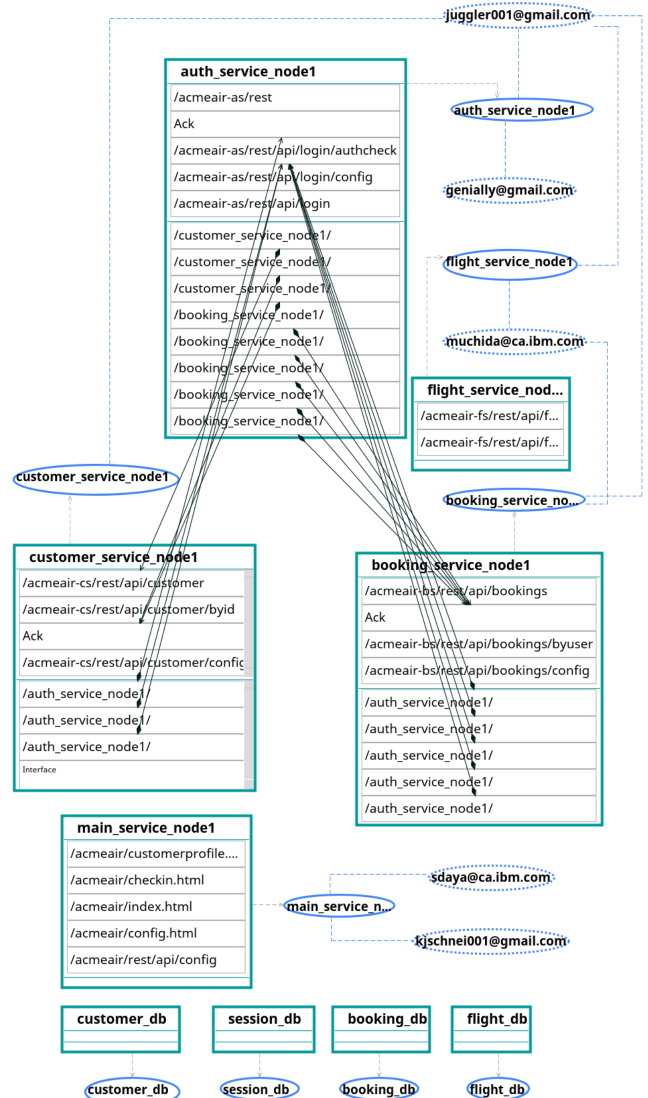
The model extracted in Figure 6a represents the Acme Air *Product*. Ten *MicroServices* have been identified by MicroART, and they are represented as rectangles with green edges. Within each *MicroService*, the *Interfaces* used for communication are represented with gray-edges boxes. The set of interfaces of each microservice is divided in two parts by a thin green line. On the top part the *exposed* interfaces are reported, while in the bottom part the *required* interfaces are shown. *Teams* are connected to the respective microservices, and are represented with blue ellipses. Similarly, *Developers* are displayed as dotted-blue ellipses and are connected to the respective teams. Due to the lack of space in Figure 6a, we have reported graphically Teams and Developers only in Figure 6b. In 6a, wrapped in a red box there is the *Nginx* service, which in the Acme Air system acts as a service discovery service. As it is clearly evident in the architectural model, the number of Links towards the service discovery is predominant.

The *architecture refinement* phase (B) allows the software architect to perform the *service discovery resolution* refinement where he selects the service discovery microservice on the *architecture model*, in order to discover the real microservices dependencies. The *architecture model* is thus refined into a *refined architecture model*, as reported in Figure 6b. In this refined model, the service discovery service has been removed and the number of dependencies has significantly been reduced, as anticipated in section II.

In the refined model, in Figure 6b, it is now possible to graphically display the real dependencies among microservices from a development perspective. In the model there are now nine *MicroServices* since the *Nginx* service discovery has been removed. The number of *Links* is significantly decreased and, most importantly, the *Links* are now showing the actual dependencies from the development perspective. Indeed, it is now possible to spot how the Main_service and Flight_service are not depending on other microservices. Indeed, while the Main_service deals exclusively with providing to the users the web page information and has no real dependencies with other microservices, the isolation of Flight_service might reveal a lack in the monitoring stage, which has not generated enough logs of the information. For this reason it is important to monitor the system for an amount of time representative of the system usage in production. Similarly, it is expected to see that many dependencies are with the Authentication_service, as users need to be authenticated in the Acme Air system to perform many of its operations. *Teams* and the *Developers* are shown respectively in blue ellipses and dotted-blue ellipses. Analyzing the distribution of developers and the size of the teams can be helpful for many reasons; for example, this information may help in identifying management issues (e.g., too many developer assigned to a



(a) *Architecture model*



(b) *Refined architecture model*

Fig. 6: Acme Air architectural models

microservice, or vice-versa) or to balance the effort distribution among microservices.

The *architecture refinement* phase provides to the software architect the possibility to reason, analyse, and refine the architecture model, for instance for performing change impact analysis at the architectural level or having a deep understanding of the overall architecture of the system.

V. Tool Support

For supporting the approach presented in Section II, we developed a tool named MicroART in order to guide the architecture recovery of microservice-based systems. This tool first extracts the information from the given repository then it generates an architecture model of the system. This model is then refined by MicroART into a *refined architecture model* by application of the *service discovery resolution* refinement.

MicroART has been realized using Model-Driven Engineering tools and development principles [7], working with model-based representations of the microservice architectures, and the tool is publicly available for download[6]. The MicroART tool has been developed on top of Eclipse and in particular the Eclipse Modeling Platform (EMF) in combination with the Spring Framework[7], as can be seen in the bottom layer of Figure 7. The *Extraction Layer* is composed of a *Repository Analyzer*, used for connecting and extracting information from GitHub repositories. The *Dynamic Analyzer* extracts infor-

[6]https://github.com/microart/microART-Tool

[7]https://projects.spring.io/spring-framework/

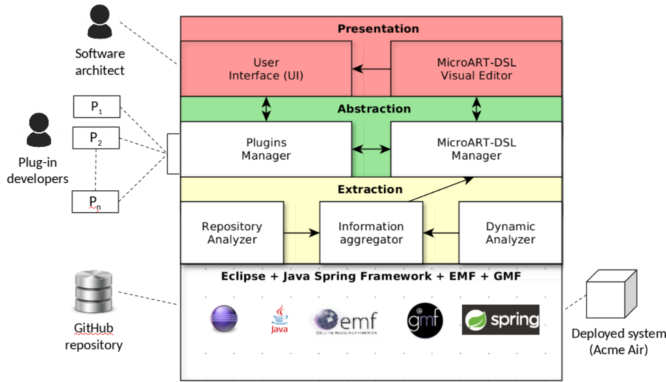


Fig. 7: Tool support for recovering microservice architectures

mation from the running system, e.g., Acme Air, and with the *Information Aggregator* composes the information to be mapped into the MicroART-DSL concepts. This translation is done by the *DSL Manager* in the *Abstraction Layer* that creates the architecture models to be rendered with the *DSL Visual Editor* developed and part of the *Presentation Layer*, built on top of the *GMF* tool[8]. The DSL manager communicates also with the *Plug-in Manager* that allows the *Developers* to integrate their resolution plugin, like the one presented in this paper, i.e., *Service Discovery Resolution* plug-in. The architect can work with the *UI* in order to manipulate the models and obtain the refined model version we described in section II. We outline that the MicroART tool architecture is extensible and it will be improved in the future to manage also other resolution patterns and increase the level of automation in the refinement phase, when allowed. Currently, the MicroART prototype implementation depends on GitHub repositories that use Docker as container engine, and thus MicroART applicability is restricted to projects based on these technologies.

VI. Related Work

The Architecture Reconstruction and MINing (ARMIN) is a tool to reconstruct deployment architectures from the source code presented in [16]. ARMIN starts the recovery process with the source information extraction, where a set of elements and relations are extracted from the system. By using the elements different views of the system's architecture are generated. The ARMIN approach differs from our because they consider only the source code extraction without considering the dynamic analysis.

Several architecture recovery techniques and tools are presented in [17]. They provide different recovery approaches and give an overview of the state of the art of this field. A framework called Mitre is introduced, and it consists of three components: the architectural representation, the source code recognition engine and ”bird's-eye” program-overview capability. The Mitre's approach is based on an abstract syntax tree. Compared to our tool despite some step of the recovery approach of Mitre are similar, Mitre is not specific for microservice based system. Also the authors present Architecture Reconstruction Method (ARM), a recovery technique that works in four major phases: development of a concrete pattern-recognition plan, extraction of a source mode, detection and evaluation of pattern instances, reconstruction and analysis of the architecture. Differently from our tool, ARM proposes a method that is aimed at systems developed with design patterns, MicroART has not this kind of limitation. Another techniques is the software architecture reconstruction (SAR) method that is based on a relation partition algebra. This method employs five levels of architecture reconstruction: initial, described, redefined, managed, and optimized. Differently to our technique, which aims to recover the dependency of the system, SAR is oriented to a system information extraction.

A SOA-oriented architecture recovery process is presented in [6]. Similarly to our approach, it is based on both a static and dynamic phases and it uses a set of tools, relying on UML to understand the system details.

In [23] the authors define an ADL by introducing a UML profile that facilitates the incremental integration specification. The approach allows developers to specify and design microservice integration, and provide mechanisms with which to automatically obtain the implementation code for business logic and interoperation among microservices. The approach is generative and differently in our approach we focus on recovering the microservices architecture.

Another work has been considered concerning DSL implementation for architectural description [12]. This approach supports full traceability between source code elements and architectural abstractions, and allows software architects to compare different versions of the generated UML model with each other. The focus is on filling the gap between the design and the implementation of a software system. Differently, in our approach we recover the microservices architecture to overcome the maintenance problem.

VII. Conclusions and Future Work

In this paper we presented an approach for semi-automatically recovering the architecture of microservice-based systems. The approach is based on MDE principles and is composed of two main steps: the first aims at *recovering* the deployment architecture of the system, while the second step has the goal to *refine* the obtained architecture. The considered architecture models conform to a dedicated DSL. The approach and the tool have been successfully applied to a third-party benchmark system called Acme Air.

Future work includes the definition of new plugins related to the component resolution functionality, such as those for resolving third-party data stores, load balancers, logging services. Also, we are investigating on identifying and defining a set of metrics for automatically evaluating key aspects of the system, such as the coupling among microservices within the system, their cohesion, their evolvability. Those metrics will live in a shared ecosystem, where third-party actors can reuse and execute them on specific microservice-based

[8]https://www.eclipse.org/modeling/gmp

systems, and even aggregate them in order to create new ones. In this context, we are planning to extend the proposed approach to support the incremental application of the metrics at run-time, during the whole lifetime of the system; software architects will have a better guidance with respect to the evolution of the system, e.g., in terms of evolution effort estimation. Finally, we are planning to extend MicroART to support components deployed in the public cloud or in other deployment platforms (e.g., Vagrant), with additional logging and rendering tools, and to investigate the application of MicroART to other microservice technologies (e.g., AWS Lambda serverless Function-as-a-Service). Additional aspects that we are considering are related to data contracts, message exchange patterns and formats, protocols, and integration with DevOps provisioning tools.

Acknowledgement

This research has been supported by the European Union's H2020 Programme under grant agreement number 644178 (project CHOReVOLUTION - Automated Synthesis of Dynamic and Secured Choreographies for the Future Internet), and by the Ministry of Economy and Finance, Cipe resolution n. 135/2012 (project INCIPICT - INnovating CIty Planning through Information and Communication Technologies).

References

[1] N. Alshuqayran, N. Ali, and R. Evans. A systematic mapping study in microservice architecture. In *Service-Oriented Computing and Applications (SOCA), 2016 IEEE 9th International Conference on*, pages 44–51. IEEE, 2016.

[2] T. J. Biggerstaff. Design recovery for maintenance and reuse. *Computer*, 22(7):36–49, 1989.

[3] H. Brunelire, J. Cabot, G. Dup, and F. Madiot. Modisco: A model driven reverse engineering framework. *Information and Software Technology*, 56(8):1012 – 1032, 2014.

[4] E. J. Chikofsky and J. H. Cross. Reverse engineering and design recovery: A taxonomy. *IEEE software*, 7(1):13–17, 1990.

[5] A. Cicchetti, D. Di Ruscio, L. Iovino, and A. Pierantonio. Managing the evolution of data-intensive web applications by model-driven techniques. *Software & Systems Modeling*, 12(1):53–83, 2013.

[6] F. Cuadrado, B. García, J. C. Dueñas, and H. A. Parada. A case study on software evolution towards service-oriented architecture. In *Advanced Information Networking and Applications-Workshops, 2008. AINAW 2008. 22nd International Conference on*, pages 1399–1404. IEEE, 2008.

[7] A. R. da Silva. Model-driven engineering: A survey supported by the unified conceptual model. *Computer Languages, Systems and Structures*, 43:139 – 155, 2015.

[8] P. Di Francesco, P. Lago, and I. Malavolta. Research on Architecting Microservices: trends, Focus, and Potential for Industrial Adoption. *IEEE International Conference on Software Architecture (ICSA)*, 2017.

[9] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina. Microservices: yesterday, today, and tomorrow. *arXiv preprint arXiv:1606.04036*, 2016.

[10] M. Fowler and J. Lewis. Microservices a definition of this new architectural term. *URL: http://martinfowler.com/articles/microservices.html*, Last accessed: Mar 2017.

[11] G. Granchelli, M. Cardarelli, P. Di Francesco, I. Malavolta, L. Iovino, and A. Di Salle. MicroART: A Software Architecture Recovery Tool for Maintaining Microservice-based Systems. *IEEE International Conference on Software Architecture (ICSA)*, 2017.

[12] T. Haitzer and U. Zdun. Dsl-based support for semi-automated architectural component model abstraction throughout the software lifecycle. In *Proceedings of the 8th International ACM SIGSOFT Conference on Quality of Software Architectures*, QoSA '12, pages 61–70, New York, NY, USA, 2012. ACM.

[13] R. L. Krikhaar. Reverse architecting approach for complex systems. In *Software Maintenance, 1997. Proceedings., International Conference on*, pages 4–11. IEEE, 1997.

[14] H. A. Müller, S. R. Tilley, and K. Wong. Understanding software systems using reverse engineering technology perspectives from the rigi project. In *Proceedings of the 1993 conference of the Centre for Advanced Studies on Collaborative research: software engineering-Volume 1*, pages 217–226. IBM Press, 1993.

[15] S. Newman. *Building microservices*. " O'Reilly Media, Inc.", 2015.

[16] L. O'Brien and C. Stoermer. Architecture reconstruction case study. 2003.

[17] L. O'Brien, C. Stoermer, and C. Verhoef. Software architecture reconstruction: Practice needs and current approaches. Technical report, DTIC Document, 2002.

[18] C. Pahl and P. Jamshidi. Microservices: A systematic mapping study. In *Proceedings of the 6th International Conference on Cloud Computing and Services Science*, pages 137–146, 2016.

[19] I. Pashov and M. Riebisch. Using feature modeling for program comprehension and software architecture recovery. In *Engineering of Computer-Based Systems, 2004. Proceedings. 11th IEEE International Conference and Workshop on the*, pages 406–417. IEEE, 2004.

[20] M. Richards. Microservices vs. service-oriented architecture, 2015.

[21] D. C. Schmidt. Guest editor's introduction: Model-driven engineering. *Computer*, 39(2):25–31, Feb 2006.

[22] C. Stringfellow, C. Amory, D. Potnuri, A. Andrews, and M. Georg. Comparison of software architecture reverse engineering methods. *Information and Software Technology*, 48(7):484 – 497, 2006.

[23] M. Zúñiga-Prieto, E. Insfran, S. Abrahao, and C. Cano-Genoves. Incremental integration of microservices in cloud applications. 2016.