Identifying Domain-Based Cyclic Dependencies in Microservice APIs Using Source Code Detectors

Patric Genfer and Uwe Zdun

University of Vienna, Faculty of Computer Science, Research Group Software Architecture, {patric.genfer|uwe.zdun}@univie.ac.at

Abstract. Isolation, autonomy, and loose coupling are critical success factors of microservice architectures. Unfortunately, systems tend to become strongly coupled over time, sometimes even exhibiting cyclic communication chains, making the individual deployment of services challenging. Such chains are highly problematic when strongly coupled communication e.g. based on synchronous invocations is used, but also create complexity and maintenance issues in more loosely coupled asynchronous or event-based communication. Here, cycles only manifest on a conceptual or domain level, making them hard to track for algorithms that rely solely on static analysis. Accordingly, previous attempts to detect cycles either focused on synchronous communication or had to collect additional runtime data, which can be costly and time-consuming. We suggest a novel approach for identifying and evaluating domain-based cyclic dependencies in microservice systems based on modular, reusable source code detectors. Based on the architecture model reconstructed by the detectors, we derived a set of architectural metrics for detecting and classifying domain-based cyclical dependencies. By conducting two case studies on open-source microservice architectures, we validated the feasibility and applicability of our approach.

Keywords: Microservice API · domain-based cyclic dependencies · metrics · source code detectors.

1 Introduction

One of the main goals of microservices is to reduce the complexity of large monolithic applications by splitting them up into smaller, autonomously acting services [30], each of them focused on a specific part of a (business) domain [4] and independently deployable [19]. In addition to being isolated from each other (and therefore also becoming more autonomous), lightweight inter-service communication is central in microservice architectures [20]. The goal is to support loose coupling of the services. Finding a balance between service isolation and interaction is often a challenging task.

Communication dependencies are often problematic when they form cycles, where a chain of service calls ends in the same service where it began [27]. As cycles increase the coupling between individual services, their independent deployment is no longer possible [6, 33]. Cyclic communication paths make the system more complex, which might also lead to higher maintenance and development efforts and costs in the long run [17].

Several solutions for this problem have been suggested, all with their specific advantages or disadvantages: Applying the *API Gateway* pattern [28], as recommended by Taibi and Lenarduzzi [27], is a possible way of decoupling the system, switching to an asynchronous communication model is another [7, 19]. While the first approach reduces the coupling, it bears the risk of creating a single point of failure in the system [15]. Worse, in this solution there is still a cycle in the system, but just via the API gateway. Changing the communication flow to asynchronous messaging is also debated among researchers. While it certainly improves the testability of the overall system as parts of the communication can easily be replaced by using message stubs, asynchronous communication makes the system also more complex and difficult to reason about [20]. Wolff [33] even argues in this context that switching from synchronous to asynchronous communication alone does not resolve any cyclic dependencies, but instead only shifts them to a different level. Before the transformation, cycles manifested through direct synchronous communication links, such as HTTP REST calls or Remote Procedure Calls (*RPC*), and were easily recognizable in the infrastructure code. But using asynchronous communication flow, these dependencies are expressed implicitly through the semantic information stored in the content of asynchronously transferred messages. For this Wolff coined the term *Unintended Domain-Based Dependencies* [33]. These dependencies – and especially the cycles resulting from them – now manifest as part of the domain or business logic and can only be resolved by redesigning the architecture. Especially tracking these communication links is now even more complicated. On the one hand, conventional metrics such as *cohesion* or *coupling* cannot correctly capture them, and most static code analysis approaches also fall short in detecting them [33].

Some authors (see e.g. [15, 33]) suggest that the only effective way to resolve these domain-based communication cycles is redesigning the architecture on the domain level. However, for this, the cyclic connections first have to be identified, which is not always a trivial task, considering the polyglot nature of microservice implementations [26]. Besides, the communication flow within a microservice system is often distributed over several endpoints, which makes tracking the communication paths difficult. Another caveat lies in the fact that those architectures have often evolved organically over time and their documentation is not up-to-date, thus making architectural reconstruction tedious. The focus on research so far has mainly been on recognizing cycles based on synchronous connections [17, 32]. Tracking and analyzing asynchronous communication on API operation level, as it would be necessary to identify domain-based cycles, has only been occasionally the subject of research so far (see e.g. [5, 12]). This work aims at filling this gap by presenting a novel approach for identifying both, technical and domain-based cyclic dependencies on microservice API operation level. To achieve this, our approach uses modular, reusable source code parsers, called *detectors* [21] for reverse engineering a communication model from an underlying microservice's source code. Based on this model, we define a set of architectural metrics for detecting and evaluating potential cyclic dependency structures within the architecture. In this context, we will study the following research questions:

RQ1 How is it possible to identify domain-based cyclic dependencies between microservices by only analyzing static source code artifacts?

RQ2 What is a minimal communication model need for tracking cyclic dependencies on API operation level?

RQ3 Can architectural metrics for supporting software architects in redesigning domain-based cycles be defined based on this model?

The structure of this paper is as follows: Section 2 compares to related works. Next, Section 3 explains how our model of the inter-service communication is generated with the help of our source code detectors. Based on this communication model, Section 4 defines various metrics for tracking and evaluating technical and domain-based communication cycles in a given microservice system. Section 5 subsequently applies these metrics on two open source example projects as part of two case studies. The remaining Sections 6, 7 and 8 conclude with a discussion of the research results, threat to validity, and future work.

2 Related Work

Cyclic dependencies are not a new phenomenon and have to be taken into account not only in microservice systems but also in monolithic applications [14]. However, their relevance for microservice systems has also been recognized in recent years by various authors [6, 17, 27, 33]. In particular, Wolff [33] distinguishes between *technical* and *domain-based* dependencies, with the latter describing dependencies that exist on a conceptual level and are difficult to track trough static analysis methods. Ma et al. [17] classify synchronous cyclic dependencies into strong and weak cycles: Strong cycles are communication paths with the exact same start and endpoint, while weak cycles end in the same microservice, but at least at a different endpoint. According to the authors, strong cycles are way more problematic as they bear the risk of potential deadlocks in case the cycle has no termination condition. Our work adopts this concept of strong and weak cycles, and extends the analyses also to asynchronous communication models.

To be able to analyze a microservice system in detail, many studies follow the approach of reverse engineering an architecture model from existing artifacts, such as architecture diagrams [18], source code or documentation [17, 22, 32], as well as Docker and other configuration files [8, 22]. In addition, runtime data such as log files or monitoring data [8, 12] is utilized for architecture reconstruction. Especially the methods that focus on source code analysis have to make some compromises regarding the used technology stack and communication protocols to reduce the parsing effort. For example, some works [17, 32] focus on Java Spring technologies and HTTP-based REST communication. In contrast, the architecture reconstruction method proposed in this work relies on lightweight, modular, and reusable source code detectors [21] that are not restricted to any specific programming language or framework.

Defining a specific set of metrics to assess the quality of a microservice architecture is also the subject of some studies: Zdun et al. [35] for instance describe metrics covering the decomposition of microservice systems, whereas Selmadji et al. [25] define a metric set for supporting the transformation of a monolithic application into several microservices. A more visual approach is proposed by Engel et al. [5]: In their study, they introduce a graph-based visual representation of their service metrics along with visualizations of metrics and their severity.

3 Static Analysis

3.1 Microservice API Communication Model

To model the communication flow observable at the API level, this paper uses a directed graph-based approach, similar to [23, 35]. We define a microservice system as a graph $G = (V, E, F)$ where V is a set of vertexes (nodes) $V = V^{ms} \cup V^{interface} \cup V^{api} \cup V^{con} \cup V^{other}$, with each subset representing one category of vertexes. E are edges of the form (v_i, v_j) with $v_i, v_j \in V$. $F = \{ms, sync, async\}$ represents a set of additional functions and predicates that operate on the graph. The different vertex categories are:

- $\boldsymbol{V^{ms}}$ represents the **Microservices**, i.e. is the set of all microservice root nodes in the system.
- $\boldsymbol{V^{interface}}$ represents the **API Interfaces**. These nodes are part of the public interface of a microservice and provide access to a specific group of (business) functionalities [16]. While they are not technically necessary for modeling the flow of communication in our model, they improve model understandability by adding more structure to the model.
- $\boldsymbol{V^{api}}$ represents the **API Operations** which play a crucial role in our communication model. They are the only direct access points to the underlying microservice functionality, making them the origin of any communication path through the system. API operations can either be synchronous or asynchronous, and have a unique address under which they can be reached, e.g., an HTTP endpoint, a queue/topic name, a specific message type, and so on.
- $\boldsymbol{V^{con}}$ represents the **Connection Operations**. The invocation of an API operation is expressed by connection operation nodes. Each of these nodes represents a technology-specific invocation of an API, e.g. by calling an HTTP endpoint or sending a message via a message broker.
- $\boldsymbol{V^{other}}$ represents any **Other Operations** in the graph which are not covered by the previous sets, such as calls to business services or domain entities.

The Set F provides the following functions and predicates:

- **ms** is a function $v \longrightarrow v^{ms}$ with $v \in V, v^{ms} \in V^{ms}$. It returns the corresponding microservice vertex when given a vertex of an arbitrary type.

That is, it returns the microservice the vertex is part of. The boundaries of a microservice end at its connection operations.

- **async** is a predicate of the form $v^{api\_con} \longrightarrow boolean$ with $v^{api\_conn} \in V^{api} \vee v^{api\_conn} \in V^{con}$ that returns *true* in case the input node uses an asynchronous communication model, otherwise returns $false$.
- **sync**: $v^{api\_con} \longrightarrow boolean$ with $v^{api\_conn} \in V^{api} \vee v^{api\_conn} \in V^{con}$ returns *true* if the given node uses synchronous communication, otherwise $false$. It can also be defined as $\neg async$.

Figure 1 shows our model as a UML meta-model.

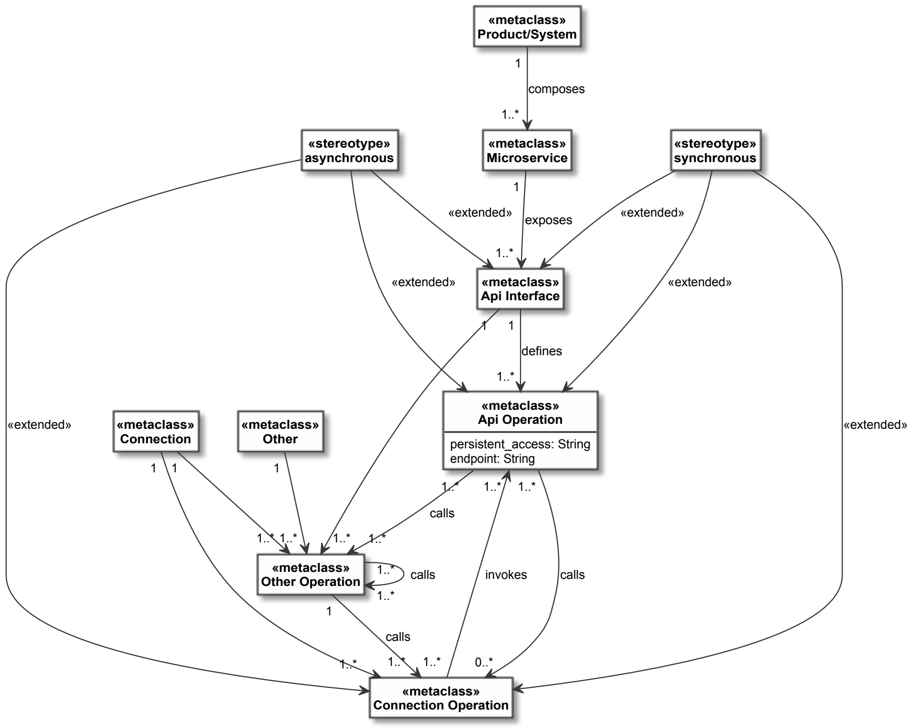


Fig. 1: Meta-model describing communication flow in a microservice system

The following constraints must be met to consider a microservice communication model as sound concerning our definition:

C1 Matching Connection Types: $\forall(v_i, v_j)$ with $v_i \in V^{con}, v_j \in V^{api}$: $(sync(v_i) \iff sync(v_j)) \vee (async(v_i) \iff async(v_j))$. This ensures that an API operation can only be called by a connection operation with a compatible communication model.

C2 A Microservice should not call its own API: $\forall(v_i, v_j)$ with $v_i \in V^{con}, v_j \in V^{api}$: $ms(v_i) \neq ms(v_j)$. Since our model focuses only on the

API perspective, we do not include the case where microservices call themselves as we consider those calls as internal system calls, not API calls, and therefore not relevant for our considerations.

C3 At least one operation per service: Since our model maps the communication on API level, we assume that a service must provide at least one publicly available API to be part of our model.

Compared to other approaches (such as [8, 13]) our model has a clear communication-centric focus. Thus, we consider connection operations and the API operations they target as central elements for describing the communication flow.

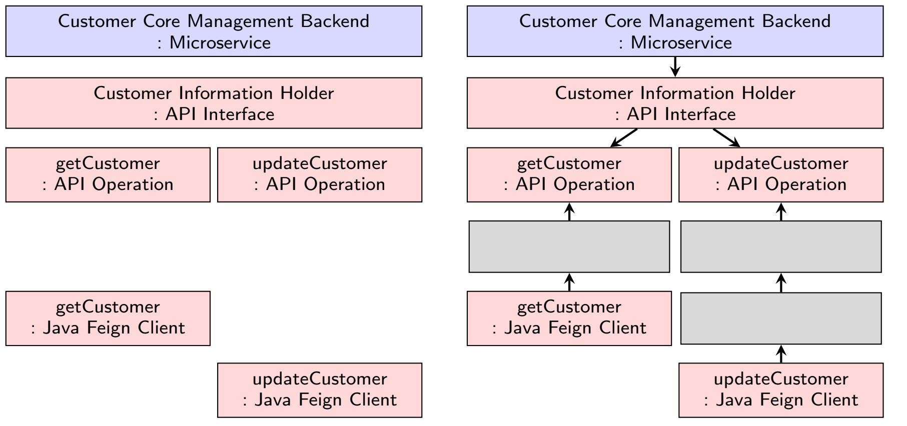
3.2 Model Reconstruction

Due to the highly polyglot nature of microservices [20], using full-blown language parsers for reconstructing the communication model out of the underlying source code would not be a practical solution. Configuring each parser and keeping it up to date would require a considerable amount of work and a deeper understanding of each language structure. Instead, our approach uses a concept from our earlier research, called modular, reusable *source code detectors* [21]. These lightweight source parsers are based on the Python module *PyParsing*[1] and scan the code for predefined patterns, while at the same time ignoring all other source code artifacts unrelated to the communication model. While these detectors must still be adopted to identify technology-specific patterns, implementing and especially maintaining them requires less effort than comparable approaches.

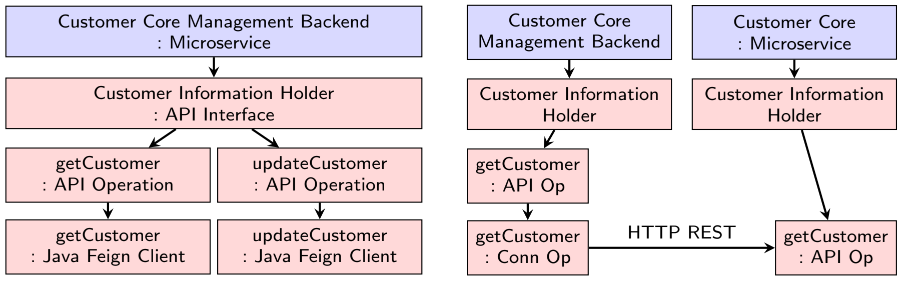
Figure 2 illustrates the four-phased reconstruction process based on an example microservice taken from the *Lakeside Mutual*[2] project. This open-source project is also used later during the case studies (see Section 5). In Phase 1, the detectors isolate all relevant model elements (also called *Hot Spots*) by scanning the source code for concrete keywords and patterns. Phase 2 uses a bottom-up search to establish invocation links between the various hot spots. During this search, the algorithm also identifies additional classes and methods or function calls as part of the invocation paths and adds them to the model as nodes of type V^{other}. Top-down connections between microservices, API interfaces, and their API operations can be created directly since, for them, no bottom-up search is necessary. After the invocation call tree is reconstructed, a path reduction algorithm again removes all non-hotspots from the model in Phase 3. Removing these elements simplifies the model drastically without losing any communication-relevant information. While this step is not necessary from a technical point of view, it significantly improves the human readability of the model. Phase 4 finally connects the isolated microservice sub-graphs by mapping the endpoint information stored in the connection operations to the corresponding API operation endpoints. This happens by matching HTTP addresses used for synchronous connections or mapping publishers and subscribers by their message types in asynchronous channels. The latter is the core focus of our work.

[1] `https://github.com/pyparsing/pyparsing/`

[2] `https://github.com/Microservice-API-Patterns/LakesideMutual`


Customer Core Management Backend : Microservice
Customer Information Holder : API Interface
getCustomer : API Operation
updateCustomer : API Operation
getCustomer : Java Feign Client
updateCustomer : Java Feign Client

(a) Phase 1: Hotspot Detection (b) Phase 2: Call Graph Construction


Customer Core Management Backend : Microservice
Customer Information Holder : API Interface
getCustomer : API Operation
updateCustomer : API Operation
getCustomer : Java Feign Client
updateCustomer : Java Feign Client
Customer Core Management Backend
Customer Core : Microservice
Customer Information Holder
getCustomer : API Op
getCustomer : Conn Op
HTTP REST
getCustomer : API Op

(c) Phase 3: Path Reduction (d) Phase 4: Interconnection Model

Fig. 2: Process for reconstructing a microservice system communication model

4 Metrics

Based on our formal communication model introduced in Section 3.1, we define a set of architectural metrics that allow us to identify and assess cyclic dependencies within a microservice system. Using our definition of a microservice system as a directed graph V, any communication chain through this system can be considered a path $p_{vu} = (v = v_1, v_2, \ldots, v_n = u)$ such that $(v_i, v_{i+1}) \in E$ for $1 \leq i < n$ [10] and $v, u \in V^{api}$. We furthermore specify $P(v)$ as the set of all outgoing communication paths *starting in* v. According to Ma et al. [17], a communication path forms a cycle whenever its start and endpoint are identical or when both endpoints belong to the same microservice. While the first case results in strong cycles, which are viewed as highly problematic, the latter results in weak cycles, which still can negatively affect the deployability of the system [17]. This leads already to the first two boolean metrics that operate on a given communication path $p \in P(v)$:

$$
\begin{aligned}
cyc_{strong}(v,p) &= \begin{cases} true, & \text{if } v_1 = v_n \quad with\, p = (v_1, v_2, \ldots, v_n), p \in P(v) \\ false, & \text{otherwise} \end{cases} \\
cyc_{weak}(v,p) &= \begin{cases} true, & \text{if } ms(v_1) = ms(v_n) \quad with\, p = (v_1, v_2, \ldots, v_n), p \in P(v) \\ false, & \text{otherwise} \end{cases}
\end{aligned}
\tag{1}
$$

These two metrics return *true* in case the communication path is either a strong cycle (cyc_{strong}) or a weak one (cyc_{weak}). By combining both metrics, we are now able to decide whether a given communication path is cyclic at all, expressed by the following boolean metric:

$$cycle(v, p) = cyc_{strong}(v, p) \vee cyc_{weak}(v, p) \tag{2}$$

By applying this metric on all outgoing communication paths of a given API operation v, one can now calculate the ratio of cyclic dependencies for this specific API operation as follows:

$$cycRatio_{api}(v) = \frac{|\{p \in P(v) : cycle(p)\}|}{|P(v)|} \tag{3}$$

Values larger than zero indicate that communication paths initiated by v result in at least one cyclic dependency with another microservice. Further investigation may be needed to determine whether this behavior is intended by design. There exists also a scenario where $P(v) = 0$, meaning that API operation v does not call any other microservices, and accordingly, has no outgoing communication paths. Then the node v is not relevant, and the metric returns 0.

Another essential aspect and one of the main contributions of this paper is the possibility to distinguish the nature of cycles further into technical and domain-based ones. For this, we first define the set of all connection operations within a given path p as $C(p) = \{c \in p : c \in V^{con}\}$. Applying the predicate *async* (see Section 3.1) on every element in C, we can determine whether a cycle exists rather on a domain or technical level.

$$domainCycRatio(p) = \frac{|\{c \in C(p) : async(c)\}|}{|C(p)|} \tag{4}$$

Since a communication path p must always have at least one connection operation, this metric is always valid. The counterpart of this metric that measures the ratio of technical connections in the cycle can trivially be expressed by:

$$techCycRatio(p) = 1 - domainCycRatio(p) \tag{5}$$

Specifying which type of cycle is less problematic is challenging to decide: While in the literature, asynchronous techniques are often recommended over synchronous ones [7, 33], scenarios with harder time constraints might require a synchronous communication flow [9]. In practice, it might also happen that an unintended cycle contains both synchronous and asynchronous service calls.

The last metric introduced here measures how often a specific API operation is part of a cyclic communication chain without being the actual root of that chain. Nodes through which disproportionately high traffic is routed are also known as hubs and play a central role in a network's topology [1]. Due to their importance, they can also have a significant impact on the creation of cyclic dependencies. Let P_{cyc} denote the set of all cyclic paths in an API model, we can then express this impact of a given vertex v through the *CyclicMembershipRatio* metric as follows:

$$cycMemberRatio(v) = \frac{|\{p = (v_1, v_2, \ldots, v_n) \in P_{cyc} : v \in p \wedge v \neq v_1\}|}{|P_{cyc}|} \quad (6)$$

Calculating this metric for a given API operation can be done by comparing the number of all cyclic paths through the given operation with the total amount of all cyclic paths in the whole microservice system.

Our current implementation calculates all of these metrics as part of a post processing step after the reconstruction process finished the model generation (see Section 3.2). Finding outgoing paths and cycles from a given API operation is achieved by using standard depth-first search and a cycle detection algorithm such as the one proposed in [29], modified to detect both strong and weak cycles.

5 Case Studies

To evaluate our approach and test the performance of our metrics, we conducted two case studies with two different open-source microservice architectures, both taken from GitHub (see footnotes below). Based on the guidelines for observational case studies suggested in [24], the source code of the projects was not altered in any way during the observation.

5.1 Case Study 1: Lakeside Mutual

Lakeside Mutual[3] models a sample microservice system that realizes the business process of a fictional insurance company (designed based on real-life insurance systems). Its level of maturity and well-documented architecture makes it a good substitute for a real production system. It is also used in several other research studies [11, 22]. The system consists of seven mostly Java Spring technologies-based backend microservices and four client applications – three Web frontends and one Node.js console client. Communication between the various services is mainly performed through synchronous HTTP REST calls and asynchronous messaging. We considered two of the seven backend services as too infrastructure-related [8] and therefore not relevant for our domain-centric approach. A third one, the *RiskManagementServer* was also not incorporated in our communication model as it does not provide any outgoing connections and hence cannot play any role in API-level communication cycles.

Our detectors were able to identify 40 API operations in total, split up into 19 API interfaces, with a majority (35) of these operations as being synchronous REST endpoints and only five asynchronous message handlers. From these 40 operations, we removed all that do not initiate a communication path, resulting in a total amount of 13 API operations for further analysis. Most of these communication paths result from the circumstance that many of Lakeside's microservices are designed as *Backend for Frontends* [3] to provide an individual

[3] `https://github.com/Microservice-API-Patterns/LakesideMutual`

interface to the underlying `Customer Core` service. Because this core service lacks any outgoing connections, none of these paths can be cyclic either.

However, our analysis found one API operation – `respondToInsuranceQuote` – with a $cycRatio_{api}$ of 0.66, meaning two-thirds of its communication chains form a cycle. Figure 3 shows the operation in question and its resulting invocation paths. Each initial invocation of this method leads to a follow-up call to the `receiveCustomerDecision` operation. From there, communication branches out in three different directions: While one path forwards to the `CoreService` and terminates there, the two others route back to the original service, resulting in two cyclic connections (see *Cycle 1* and *Cycle 2* in the diagram). Both can be considered weak cycles as each one addresses a different endpoint than the one from where the cycle started [17]. The *domCycRatio* for each of them reveals in addition that both rely on asynchronous messaging, making them purely domain-based. Resolving these cycles would therefore require a conceptual redesign of the architecture. If such a redesign is desired, the `receiveCustomerDecision` operation could be a possible starting point for further considerations: Its *cycMemberRatio* value of 1 indicates that it plays a central role in creating both cycles.

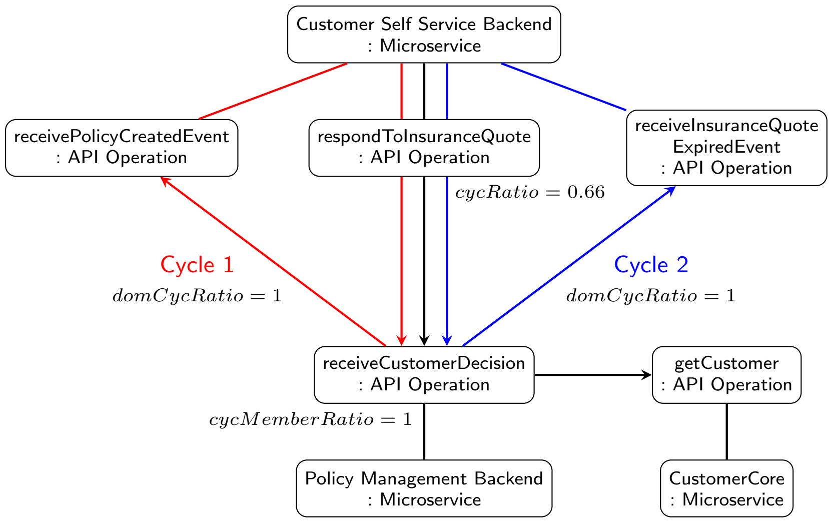

Customer Self Service Backend : Microservice
receivePolicyCreatedEvent : API Operation
respondToInsuranceQuote : API Operation
receiveInsuranceQuote ExpiredEvent : API Operation
$cycRatio = 0.66$
Cycle 1
$domCycRatio = 1$
Cycle 2
$domCycRatio = 1$
receiveCustomerDecision : API Operation
getCustomer : API Operation
$cycMemberRatio = 1$
Policy Management Backend : Microservice
CustomerCore : Microservice

Fig. 3: Communication Cycles in the Lakeside Mutual Project

We detected these cyclic relations in the *Spring Term 2020* release from March 2020 of the Lakeside Mutual Project[4], while in the most recent version, a major refactoring happened, which led to a resolution of the cycle. Whether this

[4] Available as a separate branch under `https://github.com/Microservice-API-Patterns/LakesideMutual/tree/spring-term-2020`, last accessed on June 22, 2021

was intended or just a side effect could not be determined. We checked that our tool correctly identifies in the recent release that the cycle has been removed.

5.2 Case Study 2: eShopOnContainers

The eShopOnContainers[5] repository is Microsoft's reference implementation for microservice applications and, as such, has also been the subject of various research studies ([2, 31]). It provides several frontend clients for different platforms and four backend microservices plus additional infrastructure-related services. As in the Lakeside Mutual case study, we also focus on the three backend services with the largest share of business-related domain logic. In contrast to the first study, a major difference is that all communication here is handled exclusively via an event bus (configurable to either RabbitMQ[6] or Azure Service Bus[7]). This design decision reduces the technical coupling between services to a minimum but lifts all potential cyclic dependencies to the domain level, making them harder to track. But this event-based characteristic also reflects in the implementation style of the project: The dispatch of domain messages is often decoupled from the actual creation process through event queues or caches, making the creation of an invocation graph a relatively challenging task compared to the previous case study. Our analysis revealed one cyclic dependency in the eShopOnContainers architecture (see Figure 4). Here, invoking `CheckoutAsync` operation sends a message over the event bus that is handled by the Ordering API, which sends its answer back to the Basket Service, where the responsible event handler processes it. Because of the system's asynchronous nature, these dependencies manifest obviously only on the domain level. Although this is the only cycle originating from this API operation, as can be determined by its *CycRate* metric value of 1, this link would be pretty hard to track manually since the connections between the services are not immediately visible.

6 Discussion

Regarding **RQ1**, the case studies have shown that our approach is very well suited for finding domain-based cycles. We identified this kind of cyclic dependencies in both cases by only analyzing source code artifacts without gathering time-consuming runtime information, with the second example project even using a very implicit communication model. This makes our approach particularly interesting for agile or DevOps processes, as executing our cycle checks could be done as part of the development or continuous integration pipelines. But our case study also revealed that applying our detector approach requires some additional initial effort, like identifying common technologies and coding patterns that are applied during the whole system and writing the relevant detectors to

[5] `https://github.com/dotnet-architecture/eShopOnContainers`, commit hash 6012fb... from April 12, 2021

[6] `https://www.rabbitmq.com/`

[7] `https://docs.microsoft.com/en-us/azure/service-bus-messaging/`

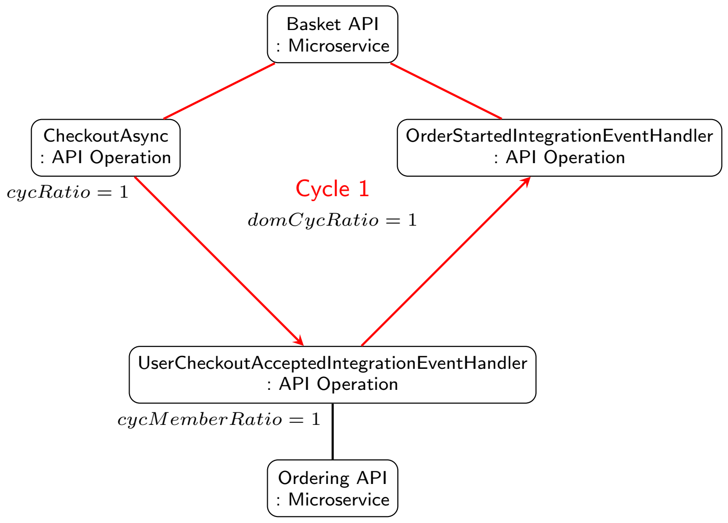


Fig. 4: Communication Cycle in eShopOnContainers

locate these patterns within the code. This upfront workload needs especially to be considered for larger systems. Complex communication scenarios, where service endpoint addresses are constructed during runtime, for example by reading input parameters, would also bring our detector approach to its limits. Here the use of additional heuristics to provide additional guidance would be necessary.

Considering **RQ2**, we could also show that our communication model contains all relevant elements to describe the information flow within a microservice system sufficiently and discover potential cycles. Focusing on API operations as central communication elements allows for a very detailed analysis of various cycle properties. At the time of this writing, we are not aware of any other research that combines so many different cycle characteristics into a single analysis.

Regarding **RQ3**, we showed in the case studies that the metrics we defined in our process provide a broad tool-set for architects to identify and assess potential cycles within a microservice application. This is especially true for domain-based cycles, which are not easily trackable as their structure often is hidden in the underlying message system. Nevertheless, the final judgment, whether a specific cyclic connection is problematic or intended, can only be made by an expert who is familiar with the underlying business domain. However, our tool-set can provide meaningful information to support a qualified decision.

7 Threats to Validity

This section gives a short overview of potential threats to validity (see e.g. [34]) and which mitigation measures we have applied:

Construct Validity expresses to what extent the correct measures were taken to study the phenomenon and how well our abstraction represents the original system. Since we developed our detectors iteratively and compared the generated model successively with the underlying source code, we can assume with a high degree of certainty that our model and the derived metrics are correct with respect to the underlying architecture. While in general possible, it is unlikely that we have missed a cycle or misinterpreted one, or made a substantial mistake in the reconstruction of the two architectures that occurred both in our manual and automatic reconstruction.

Internal Validity plays a role when there might be unknown factors that could affect the conclusions drawn from the study. Since our communication model and, therefore, our derived metrics are based on source code artifacts, all implementation related impacting factors are known at the time when the model is generated. Nevertheless, there might still exist additional artifacts like requirement specifications or specific domain knowledge that could have driven architectural design decisions. Currently, we are not considering these artifacts.

External Validity describes how well the findings could be generalized to a larger problem space and how relevant the results are beyond this specific research. The ongoing discussion about cyclic dependencies in the research community (see, for instance [17, 33] or, for real-world scenarios [6]) underlines the relevance of this problem and the example projects we used for our case studies are both open-source systems, well known to the public and research community, and combine various architectural styles and best practices. While they certainly do not reflect all possible microservice implementations, they provide a representative character to a specific extent. But still, generalization to commercial systems or systems other than enterprise domains might not be possible without adaptation of our approach.

8 Conclusions and Future Work

In this paper, we presented a novel approach for detecting technical and especially domain-based cyclic dependencies in microservice API architectures. Our approach confirms that the detection is possible by relying solely on static source code artifacts, which makes our method ideal to be applied in continuous integration pipelines. To extract our communication model from existing source code repositories, we implemented modular, reusable source code detectors and adjusted them to support different microservice systems. While this requires some upfront implementation work, our case studies revealed that this effort is manageable and can also be reduced by reusing existing detectors where possible. In the next step, we derived a set of metrics from our model, which we then used to detect and classify potential communication cycles in two open-source microservice systems during a case study. The study results show that the applied metrics can detect even inconspicuous domain-based cycles that manifest only on a conceptual level. The information gathered through our cycle analysis provides software experts with a solid foundation for making qualified decisions regarding a microservice system's architecture. While our approach can detect the existence of various types of cycles, it cannot make any assumptions about whether and how often these cycles are actually called during execution or if the deployment of the system is negatively affected through these cycles in practice. Thus, it would be necessary to collect additional runtime data and enrich the communication model with this supplementary information. We have already taken the first steps in this research direction.

Acknowledgments. This work was supported by: FWF (Austrian Science Fund) project API-ACE: I 4268; FWF (Austrian Science Fund) project IAC^2: I 4731-N; FFG (Austrian Research Promotion Agency) project DECO, no. 864707. Our work has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 952647 (AssureMOSS project).

Bibliography

[1] Al-Mutawa, H.A., Dietrich, J., Marsland, S., McCartin, C.: On the shape of circular dependencies in java programs. In: 2014 23rd Australian Software Engineering Conference, pp. 48–57, IEEE (2014)

[2] Assunção, W.K.G., Krüger, J., Mendonça, W.D.F.: Variability management meets microservices: Six challenges of re-engineering microservice-based webshops. In: Proceedings of the 24th ACM Conference on Systems and Software Product Line, pp. 1–6, ACM, Montreal Quebec Canada (Oct 2020)

[3] Brown, K., Woolf, B.: Implementation patterns for microservices architectures. In: Proceedings of the 23rd Conference on Pattern Languages of Programs, pp. 1–35 (2016)

[4] Dragoni, N., Giallorenzo, S., Lafuente, A.L., Mazzara, M., Montesi, F., Mustafin, R., Safina, L.: Microservices: Yesterday, today, and tomorrow. arXiv:1606.04036 [cs] (Apr 2017)

[5] Engel, T., Langermeier, M., Bauer, B., Hofmann, A.: Evaluation of Microservice Architectures: A Metric and Tool-Based Approach. In: Mendling, J., Mouratidis, H. (eds.) Information Systems in the Big Data Era, vol. 317, pp. 74–89, Springer International Publishing, Cham (2018), ISBN 978-3-319-92900-2 978-3-319-92901-9

[6] Esparrachiari, S., Reilly, T., Rentz, A.: Tracking and controlling microservice dependencies: Dependency management is a crucial part of system and software design. Queue **16**(4), 44–65 (2018)

[7] Garriga, M.: Towards a taxonomy of microservices architectures. In: International Conference on Software Engineering and Formal Methods, pp. 203–218, Springer (2017)

[8] Granchelli, G., Cardarelli, M., Di Francesco, P., Malavolta, I., Iovino, L., Di Salle, A.: Towards Recovering the Software Architecture of Microservice-Based Systems. In: 2017 IEEE International Conference on Software Architecture Workshops (ICSAW), pp. 46–53, IEEE, Gothenburg (Apr 2017), ISBN 978-1-5090-4793-2

[9] Hohpe, G., Woolf, B.: Enterprise integration patterns: Designing, building, and deploying messaging solutions. Addison-Wesley Professional (2004)

[10] Johnson, D.B.: Finding All the Elementary Circuits of a Directed Graph. SIAM Journal on Computing **4**(1), 77–84 (Mar 1975), ISSN 0097-5397, 1095-7111

[11] Kapferer, S., Zimmermann, O.: Domain-driven service design-context modeling, model refactoring and contract generation. Proc. of the 14th Advanced Summer School on Service-Oriented Computing (Summer-SOC'20)(to appear). Springer CCIS (2020)

[12] Kleehaus, M., Uludağ, Ö., Schäfer, P., Matthes, F.: Microlyze: a framework for recovering the software architecture in microservice-based environments. In: International Conference on Advanced Information Systems Engineering, pp. 148–162, Springer (2018)

[13] Levcovitz, A., Terra, R., Valente, M.T.: Towards a Technique for Extracting Microservices from Monolithic Enterprise Systems. arXiv:1605.03175 [cs] (May 2016)

[14] Lilienthal, C.: Sustainable Software Architecture: Analyze and Reduce Technical Debt. dpunkt. verlag (2019)

[15] Lotz, J., Vogelsang, A., Benderius, O., Berger, C.: Microservice Architectures for Advanced Driver Assistance Systems: A Case-Study. In: 2019 IEEE International Conference on Software Architecture Companion (ICSA-C), pp. 45–52, IEEE, Hamburg, Germany (Mar 2019), ISBN 978-1-72811-876-5

[16] Lübke, D., Zimmermann, O., Pautasso, C., Zdun, U., Stocker, M.: Interface evolution patterns: Balancing compatibility and extensibility across service life cycles. In: Proceedings of the 24th European Conference on Pattern Languages of Programs - EuroPLop '19, pp. 1–24, ACM Press, Irsee, Germany (2019), ISBN 978-1-4503-6206-1

[17] Ma, S.P., Fan, C.Y., Chuang, Y., Liu, I.H., Lan, C.W.: Graph-based and scenario-driven microservice analysis, retrieval, and testing. Future Generation Computer Systems **100**, 724–735 (Nov 2019), ISSN 0167739X

[18] McZara, J., Kafle, S., Shin, D.: Modeling and Analysis of Dependencies between Microservices in DevSecOps. In: 2020 IEEE International Conference on Smart Cloud (SmartCloud), pp. 140–147, IEEE, Washington DC, WA, USA (Nov 2020), ISBN 978-1-72816-547-9

[19] Nadareishvili, I., Mitra, R., McLarty, M., Amundsen, M.: Microservice architecture: aligning principles, practices, and culture. " O'Reilly Media, Inc." (2016)

[20] Newman, S.: Building Microservices: Designing Fine-Grained Systems. O'Reilly Media, Beijing Sebastopol, CA, first edition edn. (2015), ISBN 978-1-4919-5035-7

[21] Ntentos, E., Zdun, U., Plakidas, K., Meixner, S., Geiger, S.: Detector-based component model abstraction for microservice-based systems. Submitted for publication (2020)

[22] Rademacher, F., Sachweh, S., Zündorf, A.: A modeling method for systematic architecture reconstruction of microservice-based software systems. In: Enterprise, Business-Process and Information Systems Modeling, pp. 311–326, Springer (2020)

[23] Ren, Z., Wang, W., Wu, G., Gao, C., Chen, W., Wei, J., Huang, T.: Migrating Web Applications from Monolithic Structure to Microservices Architecture. In: Proceedings of the Tenth Asia-Pacific Symposium on Internetware, pp. 1–10, ACM, Beijing China (Sep 2018), ISBN 978-1-4503-6590-1

[24] Runeson, P., Höst, M.: Guidelines for conducting and reporting case study research in software engineering. Empirical Software Engineering **14**(2), 131–164 (Apr 2009), ISSN 1382-3256, 1573-7616

[25] Selmadji, A., Seriai, A.D., Bouziane, H.L., Oumarou Mahamane, R., Zaragoza, P., Dony, C.: From Monolithic Architecture Style to Microservice one Based on a Semi-Automatic Approach. In: 2020 IEEE International Conference on Software Architecture (ICSA), pp. 157–168, IEEE, Salvador, Brazil (Mar 2020), ISBN 978-1-72814-659-1

[26] Soares de Toledo, S., Martini, A., Przybyszewska, A., Sjoberg, D.I.: Architectural Technical Debt in Microservices: A Case Study in a Large Company. In: 2019 IEEE/ACM International Conference on Technical Debt (TechDebt), pp. 78–87, IEEE, Montreal, QC, Canada (May 2019), ISBN 978-1-72813-371-3

[27] Taibi, D., Lenarduzzi, V.: On the definition of microservice bad smells. IEEE software **35**(3), 56–62 (2018)

[28] Taibi, D., Lenarduzzi, V., Pahl, C.: Architectural Patterns for Microservices: A Systematic Mapping Study:. In: Proceedings of the 8th International Conference on Cloud Computing and Services Science, pp. 221–232, SCITEPRESS - Science and Technology Publications, Funchal, Madeira, Portugal (2018), ISBN 978-989-758-295-0

[29] Tarjan, R.: Depth-first search and linear graph algorithms. SIAM journal on computing **1**(2), 146–160 (1972)

[30] Thones, J.: Microservices. IEEE Software **32**(1), 116–116 (Jan 2015), ISSN 0740-7459

[31] Vural, H., Koyuncu, M.: Does Domain-Driven Design Lead to Finding the Optimal Modularity of a Microservice? IEEE Access **9**, 32721–32733 (2021), ISSN 2169-3536

[32] Walker, A., Das, D., Cerny, T.: Automated Code-Smell Detection in Microservices Through Static Analysis: A Case Study. Applied Sciences **10**(21), 7800 (Nov 2020), ISSN 2076-3417

[33] Wolff, E.: Microservices: Flexible Software Architecture. Addison-Wesley, Boston (2017), ISBN 978-0-13-460241-7

[34] Yin, R.K.: Case Study Research and Applications: Design and Methods. SAGE, Los Angeles, sixth edition edn. (2018), ISBN 978-1-5063-3616-9

[35] Zdun, U., Navarro, E., Leymann, F.: Ensuring and Assessing Architecture Conformance to Microservice Decomposition Patterns. In: Maximilien, M., Vallecillo, A., Wang, J., Oriol, M. (eds.) Service-Oriented Computing, vol. 10601, pp. 411–429, Springer International Publishing, Cham (2017), ISBN 978-3-319-69034-6 978-3-319-69035-3