Can Microservice-Based Online-Retailers be Used as an SPL?

A study of six reference architectures

Benjamin Benni
benjamin.benni@concordia.ca
Concordia University
Montréal, Québec, Canada

Sébastien Mosser
mosser.sebastien@uqam.ca
Université du Québec à Montréal
Montréal, Québec, Canada

Jean-Philippe Caissy
caissy.jean-philippe@uqam.ca
Université du Québec à Montréal
Montréal, Québec, Canada

Yann-Gaël Guéhéneuc
yann-gael.gueheneuc@concordia.ca
Concordia University
Montréal, Québec, Canada

ABSTRACT

Microservices are deployable software artifacts that combine a set of business features and expose them to other microservices. Ideally, the reuse and interchanging of microservices should be easy as they are supposed to be independent of each other, both conceptually and technologically. Selecting a service to fulfill a given feature (*e.g.*, managing a cart in a website) recalls the way *Software Product Lines* (SPL) allow variability. However, in practice, interchanging microservices requires knowing the features that the services propose, how they communicate with other services and their types. In this work, we propose to analyze service dependencies as feature dependencies, at the feature, structural, technological, and versioning level, to assess the interchangeability of services. We analyze six community-selected use-cases and report that services are non-interchangeable systematically.

CCS CONCEPTS

• **Software and its engineering → Software product lines**; **Software reverse engineering**; **Software evolution**.

KEYWORDS

Microservice Architecture, Software Composition, Reverse engineering

ACM Reference Format:
Benjamin Benni, Sébastien Mosser, Jean-Philippe Caissy, and Yann-Gaël Guéhéneuc. 2020. Can Microservice-Based Online-Retailers be Used as an SPL?: A study of six reference architectures. In *24th ACM International Systems and Software Product Line Conference (SPLC '20), October 19–23, 2020, MONTREAL, QC, Canada.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3382025.3414979

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SPLC '20, October 19–23, 2020, MONTREAL, QC, Canada
© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-7569-6/20/10…$15.00
https://doi.org/10.1145/3382025.3414979

1 INTRODUCTION

1.1 Context and Challenges

Software Product Line Engineering (SPLE) is an approach to design *Sofware Product Lines* (SPL). Their goal is to systematically reuse software artifacts by specifying their features and defining explicitly product-specific features. A feature is an "incremental unit of functionality" and features "are used to distinguish the products of a product line"[1], and the SPL manages the variability that exists when multiple features can implement the same functional requirement. For instance, an online store can *manage a cart*, but this is not a feature as we define them as *cart management* is not an *incremental* feature and does not allow to distinguish two online stores. However, *updating the quantity of a cart item* is a feature as we will use them throughout this paper.

Decomposing a software system by software artifacts, each implementing a set of features, is similar to the design and implementation of microservice-based systems. A microservice is a part of a software system and implements a set of domain-specific features, *e.g.*, managing a cart on a shopping website. A microservice is both a set of business code and the description of its deployment. In the remainder of this paper, "microservice" denotes all the code and documentation attached to the building, testing, specification, and deployment of a reachable software artifact.

Decoupling those services eases their development and reuse as they are (supposedly) independent of each other, both from a functionality and technological point-of-view. It gives much flexibility to development teams, as their technical choices (*e.g.*, programming languages) do not impact the remainder part of the system, and the service's interface is supposed to be well-documented.

However, even if microservices are meant to be *deployed* in isolation, independent from a technological point-of-view, they remain artifacts of the same final system.

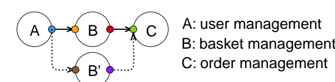

Figure 1: Simplified example of swapping microservices

For example, let us take the simplified e-commerce system described in Fig. 1. One wants to replace the cart management service `B` by another one `B'`, supposedly more efficient. Ideally, one could

take B', remove B, and make use of B'. The SPLC community identified this challenge as "Challenge 4", described as follows [3].

> **Challenge 4**: *Propose a solution that allows to interchange microservices of different technologies within a system.*

Our contribution is a methodology to assess the compatibility between services in the perspective of interchanging them. Interchanging services enables reuse of artefact and devleopement effort in an SPL-fashion. We applied our methodology to one use-case and propose a meta-analysis on other use-cases. The validation material and the information described in this paper can be found on the companion webpage on this paper [7].

1.2 Interchanging microservices

To safely interchange two microservices, they must be *compatible* at different levels that are detailed below.

Features compatibility. Compatible microservices must propose the same features. Managing a cart is not a feature *per se*, but a set of features such as adding an item, or modifying its quantity. In Fig. 1, service A makes use of a feature exposed by B; thus, B' must propose an implementation of the same feature. Such a compatibility implies that one can identify microservices feature sets. The SPL community identified this as Challenge 1 [3].

> **Challenge 1**: *Identify the features of microservice-based systems, establish a mapping to the microservices that implement them, and compare between systems*

To solve "Challenge 4", we must first address the "Challenge 1" defined above. If two services do not implement the same features set, other services must be considered to fulfill the remaining uncovered features. Composing services to cover the same feature set in case of incompleteness is out of the scope of this paper, but its results can be used towards its goal.

Communication compatibility. Microservices are communicating artifacts, and the way they exchanges information must remain the same. In Fig. 1, if one removes the service B, which is used by a service A, then the new service B' must be callable by A. Thus, we must *know* the calls that happen between services, *i.e.*, the overall system architecture to assess if previous calls remain possible.

Technological compatiblity. Microservices can be independent in their technologies (*e.g.*, libraries, languages), but the way they exchange information must be *compatible*. If one removes service B in Fig. 1, which uses *Remote Procedure Call* (RPC) to call a service C, then the new service B' must use RPC as well. If not, additional actions are needed (*e.g.*, adding a proxy between B', and C).

Code Evolution. Microservices *should* be independent in their evolution, *i.e.*, modifying one service should not imply modifying another service. This development (in)dependency is the *coevolution* of artifacts and microservices *should* not coevolve. Thus, their coevolution must be assessed. A pair of services that coevolve *may* imply a hidden dependency that must be taken into account when using one service without the other. In Fig. 1, if services B and A (and–or C) coevolve, it means that they may be coupled together and that integrating B' may add additional problems, on top of the ones already identified. The coevolution of microservices is an *indicator* that reusing one service without the other may be more difficult if they strongly coevolve than when they do not.

Proposition. To overcome Challenge 4 we identified the need to partially solve Challenge 1 first. We propose to *(i)* map services to domain features, *(ii)* establish the calls between services, *(iii)* along with the kind of technology used to exchange information, and *(iv)* an assessment of their coevolution, to assess if two services are compatible and can be exchanged with one another.

We define the notion of interchangeable microservices as follows. One can interchange two services B and B' if *(i)* they have the same feature sets, where the new service may offer additional features, *(ii)* they have compatible communication at the structural level (*i.e.*, the calls with other services are the same) *(iii)* they have compatible communication at the technological level, *(iv)* B, or B', does not coevolve with other services of its system (*e.g.*, A and C).

We make the hypothesis that a microservice implements a set of features, and we propose a methodology that considers the code, architectural, technological, and more originally, the versioning levels, to assess the level of compatibility between services. Using a development metric (*i.e.*, a codevelopment indicator) with a feature mapping and a call-graph annotated by the type of communication protocol used, we assess if two services are compatible and interchangeable. We applied our methodology on one use-case selected by the SPL community [3], and gather the results of applying our methodology on five other use-cases in the form of a meta-analysis.

First, we present the methodology in Sec. 2, before applying it to a concrete use-case in Sec. 3, and presenting a meta-analysis of our findings on six use-cases identified by the SPL community. Then, before concluding in Sec. 6, we present threats to validity in Sec. 4 and related work in Sec. 5.

2 METHODOLOGY

Thus, this section describes the overall generic methodology that we applied latter in this paper, to gather information about features, communication, and technological compatibility ; and codevelopment to assess the interchangeability of two services.

2.1 Features as a Service

The first step is to *map* features to services, allowing us to identify potential variability by identifying potentially swappable services.

This feature mapping can be defined from a feature to microservices, specifying which features are offered by a given microservice ; or the mapping can preferably be from feature to endpoint when possible. An API endpoint is a remotely callable point of a service, and is a portion of a whole API (*e.g.*, Getting a product description is likely to be implemented as an endpoint). The more fine-grained the mapping will be, the harder it will be to extract it. However, it will significantly improve the detail of knowledge that one can have about communications between services. Instead of knowing that "*service* A *calls service* B", one would benefit from knowing that "*when performing operation X in service* A*, this remotely calls the operation Y in services* B", leading to a finer-grain archicture.

Each microservice can implement various features, *e.g.*, a CartManagment service may implement an $AddItem$, $ModifyQuantity$, $RemoveItem$ feature. Depending on the data available, one can map features to services (coarse-grained), or to the specific handling

endpoint (fine-grained). Considering feature granularity is necessary, as one has no guarantee that a fine-grained feature mapping is feasible in practice due to the *intrinsic heterogeneity* that microservices development offers. Over the six microservices systems that are part of the study, we note the usage of six different languages, eleven deployment technologies, six different databases technology, six messaging technology, and two embedded dynamic trace mechanisms. These projects gathered different development teams, were developed in various ways, following their guidelines and standards, which adds to the high-level of technological heterogeneity. Thus, in the context of analyzing microservices, the quest for a fully-automated and perennial tool that can reverse-engineer *any* microservices architecture is pointless by design. On the one hand, considering the high heterogeneity in the existing technologies, and the ever-faster growing number of new frameworks, static code analysis approaches will quickly reach a limit. On the other hand, dynamic approaches (*e.g.*, , analyzing traces of execution) exist but are tied-up to the scenarios used as input. Thus, these issues prevent the definition of a silver-bullet approach that would exhaustively reverse-engineer any microservice architecture.

Therefore, a semi-automated (hence, semi-manual) extraction step is required. One can take advantage of the relative existence of standards to extract potentially useful information automatically. Automation can help by reducing the search space and speeding-up such analysis at the human scale (*e.g.*, by spotting keywords). It is also possible to leverage development frameworks. Our team is developing the Anaximander tool to support microservices' architects while exploring architectures,[1] providing static analyses used to extract information from the source code.

We first looked for structural elements that match particular regular expressions: *e.g.*, folders or filenames that end with *service*. Then, we identified if the microservice under study follows any particular standard regarding API specification. For example, the existence of a swagger's open API specification drastically reduces the search space and speeds up the analysis which is something our tool leverages. Language-specific strategies also eased the search for other projects, *e.g.*, NodeJS/Express projects often have a server.js file that describes the API endpoints using HTTP routes, whereas Java projects may use RPC in which case a folder dedicated to RPC specification is present. Anaximander can also automatically parse the deployment descriptors (e.g., Docker-compose, Kubernetes) that describe the services or leverage the internal DNS or the service registry used to support runtime service resolution even if we did not explore this path.

2.2 Services Dependencies

Now that we know the mapping between services and features, we need to establish the relationship, exchanges, and calls between services. Mapping these exchanges will allow us to chart the microservices architecture and the communication technologies used. We build a call-graph `G` which is a directed graph where nodes are services, and vertices are exchanges between those services. These exchanges can be coarse-grained, *i.e.*, establishing communications between microservices, or fine-grained, *i.e.*, establishing communications at the endpoints level. A more precise analysis will necessarily provide a more accurate map and enable more precise reasonings. However, it is less feasible *in practice*, requires a non-negligible amount of work, and faces a high heterogeneity of technologies and languages. Nevertheless, one can leverage this cartography to partially assess a given microservice's isolation, *i.e.*, the dependency of a feature regarding the others.

Coarse-grained call-graphs are already available as documentation in some of the six use-cases.[2] To the best of our knowledge, no fine-grained call-graph are available.

To build such dependencies graph one can :

- leverage standards that may be adopted by the use-case (*e.g.*, microservices-demo[3] make use of Swagger that leverages the OpenAPI standard [14],
- leverage common practices and frameworks.

Type of communication. Then, another critical aspect is the *type* `t` of a communication `C`. Asynchronous communications, via messages exchanges, are less prone to create a services-dependency as the one sending out the message does not know the recipient, if one is even available. By essence, synchronous communications that trigger a specific *function* (in a broad meaning) of a remote service, create a dependency between these services. Thus, the type `ASYNC`, `SYNC` of the communication should be retrieved as it is an essential factor that weight the communication itself. Two microservices communicating via Remote Procedure Call (RPC), HTTP REST calls, or via a message-channel, do not have the same level of (in)dependence. We argue that exchanging messages through a channel is the most relaxed dependency and the most adaptable one, as the messages can be adapted quite merely to enable microservices interoperation. In contrast, one cannot adapt REST APIs quickly, and must develop a proxy in order to enable interoperation.

2.3 Service codevelopment at the Versioning Level

Services are supposed to be developed independently, at least at the technological and feature level. However, frequent joint modifications of a set of services, called codevelopment, can occur. In real-life scenarios, services are not systematically codeveloped or never codeveloped; it is a *spectrum* which is an indicator among others of dependencies and should be considered as that: an *indicator.*

Considering a git-based versioning system, as it is the one used by the six selected use-cases, we extract information contained in *commits.* A commit is an identified set of modifications, each one targeting a specific file. We need to:

1. analyze the versioning of each system: this implies retrieving the whole versioning history, and going through commits,
2. map the commit to a set of services modified: as commits contain modifications, each one targeting a specific file, we need to assess if a given modified file is part of a given service or not,
3. keep the commits that modify at least two services,

[1] https://github.com/ace-design/anaximander-microservices

[2] GoogleCloudPlatform/microservices-demo

[3] https://github.com/microservices-demo/payment/blob/master/api-spec/payment.json

(4) perform a codevelopment analysis by computing the number of times a set of services has been modified together. This computation is a pourcentage over all commits.

A high level of codevelopment indicates that two services may have hidden dependencies; more hidden dependencies than services with low level of codevelopment.

2.4 A Metric for Interchangeability

This section leverages the elements extracted by the steps presented previously.

Given the features offered by some services, their respective calls to other services and the technology used to communicate, and more originally given their respective versioning history, we establish an interchangeability metric, where the definition of interchangeability is the one specified in Sec. 1. This composed metric is not a number but can only be a gradient that indicates if two services are *likely* to be highly interchanged or not.

Two services that offer the same features, are connected to the same services, can be called using the same technology, and that have a low codevelopment level (in their respective history) have a *high* level of interchangeability, higher than two other services that have a high level of codevelopment, and that use different technologies to communicate with the remainder of their respective systems, which will have a *low* level of interchangeability.

3 RESULTS

3.1 Applying methodology

This subsection details the application of the methodology described in the previous section on one of the use-cases, namely `GoogleCloudPlatform/microservices-demo`,[4] focusing on how *concretely* we retrieved the data, leveraged the existing standards or the technologies used. The validation material and the information described in this section can be found on the companion webpage on this paper [7].

Feature mapping. At a coarse-grained level, the mapping from feature-to-service can be quickly done by analyzing the documentation at our disposal.[5]. This documentation allows us to *(i)* list the available services in the system, *(ii)* map each service to a high-level feature, *(iii)* enable technology-specific search as the language in which each service has been developed is provided. In this use-case, no particular API-definitions standard seems to have been used. While other projects may take advantage of OpenAPI specification,[6] speeding up the whole fine-grained cartography, a more manual time-consuming approach seems to be needed here.

As it is often the case, the repository layout is helpful to navigate through services' definitions quickly. Here, a folder named `xxxservice`, under the *src* folder at the root of the repository, wraps the files related to it, where *xxx* is the service name.

At this step, analyzing a specific microservice will depend on the language and technologies used. Let us take the `checkoutservice` as an example. It is written in `Go` and does *not* define a specific communication interface. We first need to localize the *main* file, which is, as usual in a `Go` project, `main.go`. We manually browsed this file and looked for the methods' names to map them to features: *e.g.*, a `PlaceOrder` method exists in the `main.go` file; thus we assume that the service allows one to place an order.[7]

Another example is the payment service, which is written in `Javascript` in the `Node` ecosystem, using the `Express` framework. This is a common association in the JS-based web backend development. As usual, in this language, a `server.js` file defines the endpoints of the server. Here, a quick manual analysis reveals that only one endpoint (*i.e.*, one feature) is present: `charge`[8]: one can use this service to `charge` a customer. Further analysis at the function-level will point us toward the `charge.js`, which is imported by the main file. This specific file details the different features of the `charge` one,[9] for instance, which type of credit card it accepts.

Table 1 summarizes the feature mapping that we found by guided manual extractions following the described methodology.

Table 1: Features identified in the `GoogleCloudPlatform/microservices-demo` use-case

Service	Features
Ad	Get ads, random, or per category
Cart	Basic cart management
Currency	Convert and List supported currencies
Email	Send order confirmation
Payment	Charge (Visa + Mastercard)
Product Catalog	List and search for product(s)
Shipping	Get a quote or ship an order

Service communications and technologies. The documentation, again, quickly helped us to obtain a coarse-grained cartography [10]. This technology-agnostic map is useful to map the overall architecture and links between services but does not specify how they communicate. The technology used is indicated in the documentation[11] and is supposed to be Remote Procedure Call (RPC). As always, with documentation, we should be cautious of the information that we found. Analyzing each service will allow us to confirm the documentation information. RPC is a synchronous protocol in the sense that the sender must know the address of the recipient in order to trigger a procedure remotely. Analysis of services such as checkout will confirm the use of RPC, as the services used are listed, and will allow us to build a fine-grained cartography.[12]

Knowing that it is a Go service that uses RPC to communicate and that the potentially used services are listed, a textual search

[4] https://github.com/jacobkrueger/SPLC2020-Microservices-Challenge/tree/GoogleCloudPlatform/microservices-demo
[5] https://github.com/jacobkrueger/SPLC2020-Microservices-Challenge/blob/c472bd3548032403bbd92127ee98d4831bb4ab23/README.md#service-architecture
[6] https://github.com/microservices-demo/catalogue/blob/97545f4ae068250190491a970a8940c44f1cd2a5/api-spec/catalogue.json
[7] https://github.com/jacobkrueger/SPLC2020-Microservices-Challenge/blob/c472bd3548032403bbd92127ee98d4831bb4ab23/src/checkoutservice/main.go#L199
[8] https://github.com/jacobkrueger/SPLC2020-Microservices-Challenge/blob/c472bd3548032403bbd92127ee98d4831bb4ab23/src/paymentservice/server.js#L47
[9] https://github.com/jacobkrueger/SPLC2020-Microservices-Challenge/blob/c472bd3548032403bbd92127ee98d4831bb4ab23/src/paymentservice/charge.js#L71
[10] https://github.com/jacobkrueger/SPLC2020-Microservices-Challenge/blob/c472bd3548032403bbd92127ee98d4831bb4ab23/docs/img/architecture-diagram.png
[11] https://github.com/jacobkrueger/SPLC2020-Microservices-Challenge/tree/c472bd3548032403bbd92127ee98d4831bb4ab23#features
[12] https://github.com/jacobkrueger/SPLC2020-Microservices-Challenge/blob/c472bd3548032403bbd92127ee98d4831bb4ab23/src/checkoutservice/main.go#L63

using traditional tools (*i.e.*, `Ctrl+F`) will allow us to map which checkout endpoints call which other service's endpoint. Looking for `productCatalogSvcAddr` will inform us that checkout calls the products catalog service only to retrieve items' information when an order is placed;[13] that the user's cart service is called twice as the checkout service allows one to retrieve or empty a cart [14].

Codevelopment. We analyzed the versioning history of the use-case and extracted a script-friendly log to incorporate as much automation as possible. We run the command `git log –name-status –oneline` that returned each commit, and grouped the files this commit has modified on a per-line basis.

Again, using keywords (*e.g.*, names contain '`service`') and folder structures (*e.g.*, the modified file is in `./Services`), we could assess which services are modified in a given commit. This is a use-case specific and technology-specific extraction step. In the case of the `GoogleCloudPlatform/microservices-demo` project, we developed a script that automatically extracts the commits that contain modifications that target at least two different identified services. If you perform such analysis on all commits, for all services, you will have an overview of how many times a set of services have been modified together, *i.e.*, have been codeveloped.

Figure 2 is an UpSet plot [11] that displays the intersection between sets of commits where multiple services have been modified. It can be read as follows: the first column states that all services have been modified together in 3 commits (the height of the first vertical bar, at the top of the chart) and that `payment`,`recommendation`, and `checkout` has been modified in more than five commits (7th vertical bar). The horizontal bar (at the right side of the figure) shows that the shipping service is the most modified one, whereas payment is the least modified.

This visualization quickly shows that the three services `checkout`, `productCatalog`, and `shipping` are modified together more than 20 times, which is more than half the number of commits for each service (*i.e.*, 20 over 30). Moreover, `productCatalog`, and `shipping` have been modified together around 30 times, which is the number of commits that modifies those services. Those two services have a high codevelopment level, and `checkout` is also tightly codeveloped with them. According to our definition, this means that swapping one of these three services for another version will more likely be a hard and tedious process if even possible at all.

3.2 Meta-analysis

We applied the same methodology we described in the previous section to the six selected use-cases. Instead of repeating the description of the same analysis and only changing the use-case, this section describes the analysis of the results of all use-case analyses.

We analyzed the feature-mapping, the service calls and the codevelopment for each use-case on which we kept the 5th most codeveloped services. These information can be found on the companion website of this paper [7].

Once we aligned the vocabulary (*e.g.*, cart ≡ basket) and that we looked to services as a features-set, we found out that the pair

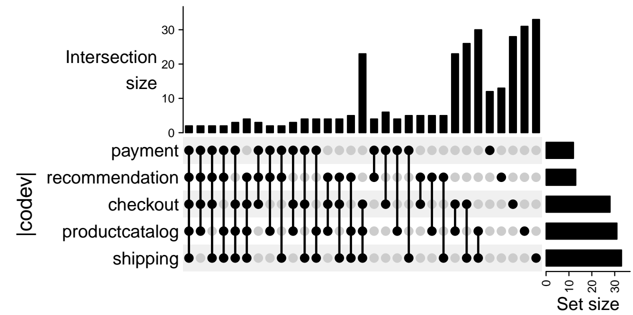

Figure 2: Codevelopment in GoogleCloudPlatform microservices-demo use-case

$(catalog, order)$ was highly codeveloped. This pair of services ranks successively 3rd, 1st, 4th, 1st, and 3rd in terms of co-developpement. Adding to this that half of the use-cases have a synchronous communication method between their catalog and order services (*i.e.*, via REST API calls, or RPC) allows us to conclude that, regardless of the use-case you chose, taking the catalog service without the order one may be a difficult and error-prone task that would most likely need additional integration steps.

The same way, when available, the cart and order services are tightly codeveloped, ranked 2nd, 1st, and 1st in the most codeveloped services in half of the use-cases. Again, they use synchronous communications in half of the use-cases leading to the same conclusion than the previous results.

Adding more automation in the process and growing the number of use-cases studied may allow us to obtain further findings that we think are of significant interest for a safe reuse purpose.

Given semi-manual and guided processes, we successfully mapped features to microservices, providing an answer to the challenge 1 identified by the SPLC community. Then, thanks to this new mapping, and with three other indicators (communication and technological compatibility, and codevelopment level) we proposed an answer to challenge 4 by assessing interchangeability of two microservices.

4 THREATS TO VALIDITY

The first threat to validity is related to the lack of validation of the results outside of the carefully picked six reference use-cases. Those use-cases all target the same business domain, lacking domain-heterogeneity, an aspect that may be interesting to consider. This would hardly be mitigated as open-source microservices may be particularly hard-to-find in the wild; as they are at the core of the business logic, companies are reluctant to share those artifacts.

Moreover, sharing the code versioning is a step further that may be hard to find in real-life use-cases. In addition to that, our approach assumes that a single code repository hosts the whole code and that all microservices are developed in this code repository. This was true for 5 out of 6 of our use-cases but is a threat to validity for our approach.

Moreover, these six use-cases that we studied were meant for demonstration purposes, where one of those would incorporate as many technologies and languages as possible on prupose; or

[13] https://github.com/jacobkrueger/SPLC2020-Microservices-Challenge/blob/c472bd3548032403bbd92127ee98d4831bb4ab23/src/checkoutservice/main.go/#L330
[14] https://github.com/jacobkrueger/SPLC2020-Microservices-Challenge/blob/c472bd3548032403bbd92127ee98d4831bb4ab23/src/checkoutservice/main.go#L301

redevelop the same task using different implementations (*e.g.*, deployment artifacts for each provider). This "show-off" effect does not reflect the actual and real way of developing microservices at the enterprise level but in itself covers a lot of variations and different scenarios in a small size of the sample.

5 RELATED WORK

Codevelopment. It is a kind of relationship that establishes that a set piece of software evolves together in time. This evolution does not take the form of strong dependency (*e.g.*, import, method call), but is a form of "soft" dependency when evolving an artifact implies a modification on the other. Work has been done toward analyzing such codevelopment, notably in the context of Linux [15], exploiting code-versioning to *understand* how variability models and related artifacts coevolve in this context. In our work, we use codevelopment factor as an indicator among others of how "easy" it may be to reuse features implementations, *i.e.*, to reuse a microservice.

Feature location. It is the activity of identifying a feature at the code-level that implements the functionality of a given system. This can be done using various techniques (*e.g.*, static analysis, semantic-based information retrieval, traces analysis), including exploiting version control systems [6], and newly analysis on object-oriented code seems to bring interesting outcomes [13]. In the case of family of systems, the feature mapping (or absence of mapping) in a given system is, most of time, known *beforehand* [12]. While state-of-the-art techniques use development history to localize and identify features, we used it to detect feature *interactions* by assessing the codevelopment of microservices.

Service extraction. Work has been done toward microservices extraction by breaking down a monolithic system into manageable microservices. The extraction process seems to work well in practice [10] but may not be well documented as how researchers actually identified those services, and that variability is "*a key criterion for structuring the microservices*" [5]. Properly identified criteria have been identified (and more and less successfully applied) to decompose those systems [8]. Reengineering legacy applications using SPL is a trend that still faces research challenges [2]. Other researchs argue that static code analysis reach a limit as to properly identified functionalities and that dynamic search are needed [9]. Human expertise is often the key to successful extraction and interviews must be lead to *first* extract this knowledge before using it [4].

6 CONCLUSION

A microservices architecture eases development as each service is developed in isolation, giving much flexibility to the development team. As a microservice implement a set of domain-features, one should be able to *reuse* a microservice, or replace it by an *equivalent* one. Those aspects bridge with the SPL research field whose goal is to organize features, and engineer features-based solutions. To assess the feasibility of service replacement, *i.e.*, feature implementation selection, we highlighted the need to assess their technical and functional compatibility and their independency regarding the other services at the code-source, and code-evolution level. The originality of our approach was to leverage *classic* semi-automated static code analysis and mixing its results with an analysis of codevelopment at the versioning level. We found out that, over six selected use-cases, two-pairs of microservices were dependent on each other, preventing any straight-forward reuse or inclusion in another software system. In future work, we plan to assess how state-of-the-art tools and approach manages the highly and intrinsic polyglots aspect of microservices architecture and development, as this seems a necessary step towards reusing state-of-practise results in the microservices architecture area.

ACKNOWLEDGMENTS

This research has been supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) and the Inria - *Équipe Associée* program (CAPESA). The authors want to thanks Florian Vouters who implemented some of the Anaximander probes used to perform the static analysis described in this paper.

REFERENCES

[1] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2013. Software Product Lines. In *Feature-Oriented Software Product Lines*. Springer, 3–15.
[2] Wesley KG Assunção, Roberto E Lopez-Herrejon, Lukas Linsbauer, Silvia R Vergilio, and Alexander Egyed. 2017. Reengineering legacy applications into software product lines: a systematic mapping. *Empirical Software Engineering* 22, 6 (2017), 2972–3016.
[3] W.K.G. Assunção, J. Krüger, and W.D.F. Mendonça. 2020. Variability Management meets Microservices: Six Challenges of Re-Engineering Microservice-Based Webshops. In *24rd International Systems and Software Product Line Conf., SPLC*. ACM, Montreal, Canada.
[4] Emil Axelsson and Erik Karlkvist. 2019. *Extracting Microservices from a Monolithic Application* . Master's thesis. Chalmers University of Technology and University of Gothenburg, Gothenburg.
[5] Luiz Carvalho, Alessandro Garcia, Wesley K. G. Assunção, Rodrigo Bonifácio, Leonardo P. Tizzei, and Thelma Elita Colanzi. 2019. Extraction of Configurable and Reusable Microservices from Legacy Systems: An Exploratory Study. In *Proceedings of the 23rd International Systems and Software Product Line Conf.* (Paris, France). Association for Computing Machinery, New York, NY, USA, 26–31.
[6] Bogdan Dit, Meghan Revelle, Malcom Gethers, and Denys Poshyvanyk. 2013. Feature location in source code: a taxonomy and survey. *Journal of Software: Evolution and Process* 25, 1 (2013), 53–95.
[7] Benni et al. 2020. Companion webpage of this paper. https://github.com/ace-design/splc2020-challenge-companion. Accessed: 2020-06-22.
[8] Michael Gysel, Lukas Kölbener, Wolfgang Giersche, and Olaf Zimmermann. 2016. Service cutter: A systematic approach to service decomposition. In *European Conf. on Service-Oriented and Cloud Computing*. Springer, 185–200.
[9] W. Jin, T. Liu, Q. Zheng, D. Cui, and Y. Cai. 2018. Functionality-Oriented Microservice Extraction Based on Execution Trace Clustering. In *2018 IEEE International Conf. on Web Services (ICWS)*. 211–218.
[10] Alessandra Levcovitz, Ricardo Terra, and Marco Tulio Valente. 2016. Towards a Technique for Extracting Microservices from Monolithic Enterprise Systems. *CoRR* abs/1605.03175 (2016). arXiv:1605.03175
[11] A. Lex, N. Gehlenborg, H. Strobelt, R. Vuillemot, and H. Pfister. 2014. UpSet: Visualization of Intersecting Sets. *IEEE Transact. on Visualization and Computer Graphics* 20, 12 (2014), 1983–1992.
[12] Jabier Martinez, Nicolas Ordoñez, Xhevahire Tërnava, Tewfik Ziadi, Jairo Aponte, Eduardo Figueiredo, and Marco Tulio Valente. 2018. Feature Location Benchmark with ArgoUML SPL. In *Proceedings of the 22nd International Systems and Software Product Line Conf. - Volume 1* (Gothenburg, Sweden) *(SPLC '18)*. Association for Computing Machinery, New York, NY, USA, 257–263.
[13] Johann Mortara, Xhevahire Tërnava, and Philippe Collet. 2020. Mapping Features to Automatically Identified Object-Oriented Variability Implementations: The Case of ArgoUML-SPL. In *Proceedings of the 14th International Working Conf. on Variability Modelling of Software-Intensive Systems* (Magdeburg, Germany) *(VAMOS '20)*. Association for Computing Machinery, New York, NY, USA, Article 20, 9 pages.
[14] OpenAPI. 2020. Open API specifications. https://swagger.io/specification/. Accessed: 2020-06-22.
[15] Leonardo Passos, Jianmei Guo, Leopoldo Teixeira, Krzysztof Czarnecki, Andrzej Wąsowski, and Paulo Borba. 2013. Coevolution of variability models and related artifacts: a case study from the Linux kernel. In *Proceedings of the 17th International Software Product Line Conf.* 91–100.