Algorithm for Detecting Antipatterns in Microservices Projetcs

Rodrigo Borges[0000−0001−6920−3576] and Tanveer Khan[0000−0001−7296−2178]

Tampere University Tampere Finland
rodrigo.borges@tuni.fi
tanveer.khan@tuni.fi

Abstract. When migrating from a monolithic to a microservice-based architecture, one need to know patterns and antipatterns in order not to propagate old practices learned from centralized systems to a new structure where services are independent and distributed. We select 5 known antipatterns in microservices-based solutions and propose an algorithm for detecting them automatically. In a first round, all classes, methods and imports are registered and associated to its location in the source code. In a second round the usage of these resources is mapped and antipatterns are detected. The algorithm is also responsible for generating visual output of how resources are used in the project, so the developers can manage how resources are distributed in different files or concentrated in few ones. The solution can avoid common mistakes when deploying microservices-based projects and can help project managers to get an overview of the system as a whole. The algorithm is tested in a well-known open source project revealing possible improvements and resource allocation information.

Keywords: Microservices · Monolithic · Network Theory · Antipatterns

1 Introduction

Microservices are now becoming a well known model for software engineers and architect. It resembles divide and conquer approach in which a monolithic software or system is decomposed into independent services [1]. Currently, microservices are adopted by big organizations such as Linked In, Amazon, Soundcloud and Netflix. These microservices are small and autonomous services [16]. All of these services work together each of which has a clear define goal [14].

The monolithic software are being in use since computers are introduced. These software are not only specific for a single task but perform each and every step required for a particular function. In these type of applications the developers have a broad access to a source code. In addition, all the dependencies that are common have a single set. In short monolithic applications are adopted because of [5]:

[1] https://itnext.io/anti-patterns-of-microservices-6e802553bd46

- Standardization: The engineers or developers are governed how to interact with the code using a shared set of tools. Such as reviewing the code, building and testing the code etc.
- Visibility: The code for monolithic softwares is available to all the developers working in a team. Every one can see the work done by the rest of team members.
- Completeness: A whole project can be built from a single repository using all of its dependencies.
- Centralization: One single repository for the whole code is available.
- Synchronization: The team works and all the commits made are synchronized.

Google still uses this monolithic architecture in which the whole code for the project is available in one single repository. This code is accessible by 20,000 engineers and consist of 2 billion lines of code [5, ?]. However, Monolithic applications are hard to scale up both organizationally and technologically as the whole team works on the same code stored in same database. The problem with these application is, a small change in any part might have impact on the other parts[2]. The redeployment of such application can take hours. Sometime for new developers, go through the whole code is not an easy task. In short, Monolithic applications are expensive, slower and even difficult for new developers to understand the whole code[2].

In order to improve the design and architect various principles are adopted. The earliest principles being adopted are Information hiding [4], encapsulation [13], separation of concerns [3], modularity [17], service oriented architecture [1] and then microservices.

Now migrating from monolithic to microservices, one cannot only face certain technology related issues but issues related to processes, strategy and organization. There are problems while migrating for which a solution is available, but that solution does not work well in these circumstances. [3]. The reuse of a software is a productive technique, having mixed success. There are a number of anti patterns emerge while analyzing the reusability failure [7]. In paper [15], the authors focused on the architectural pattern of microservices. They identify a number of agreed microservices architectural patterns. In a number of case studies these microservices architectural patterns are adopted and reported.

Some of the well known anti patterns are: [4]

- Everything Micro: Among the common anti patterns is everything micro. This type of anti pattern is common to enterprise organization. In this case one huge data store is available for all the microservices. The main challenge with this type of anti-pattern is keeping track of the data.

[2] https://itnext.io/anti-patterns-of-microservices-6e802553bd46

[3] https://microservices.io/microservices/antipatterns/-/the/series/2019/06/18/microservices-adoption-antipatterns.html

[4] https://dzone.com/articles/microservices-anti-patterns

- Break the Piggy Bank: Another most common anti pattern is break the piggy bank. When an existing application is refactoring to microservices. The refactoring is risky and can take several hours if not days.
- We are Agile: Shifting from waterfall software development to agile software development. Initially, the team start implementing a small initial version of agile-fall. In this type, it is just like merging parts together that become worst after time.

The text is structured as follows. In a first section, we present the selected anti-patterns to be detected, and a short description of how the algorithm was implemented to detect them. In the following one, we present a methodology for retrieving a general structure of a microservice-based project in terms of resources, and bring two metrics for measuring closeness and betweenness of complex networks. We then present a case study, applying the algorithm to an open source project, pointing out its anti-patterns and a general description. We conclude by evaluating the proposition and proposing some future work.

2 Anti-patterns

We have selected some of the anti-patterns from an online list[5].

- Ambiguous Service - It refers to the situation when microservices interface elements are named inappropriately [10, 9, 11]. It can occur that an operation gets to be called with a too long name, or general messages with unclear or ambiguous names. In these cases a threshold can be set for defining the maximum length of elements, and certain terms can be considered as inappropriate for a specific context.
- API Versioning - It can happen that a request to an external service is updated in the code, and that the new call points to a newer version of the API. This can promote lack of resources for the subsequent operations that depend on this data. This is why API needs semantically consistent versions descriptions (e.g. v1.1, v1.2), so developers have control of the content they dealing with [14]. Detecting bad named API versions can be quite challenging, since bad choices can have any format. We propose a simple solution that can be further developed in future works.
- Hard-Coded Endpoints - IPs addresses and ports may be hard-coded for each service [14], and this might provoke similar problems than the ones described previously. IP addresses can change from time to time, specially if they are defined dynamically by the hosting service, and fixing it would require entering files and changing it one by one. In the present work we are simply detected hard-coded IP addresses without going any further on the analysis of the context it is happening.
- Bottleneck Service - A service consist of many consumers and one point of failure [8, 11]. As this service is used by a large number of other clients

[5] https://xjreb.github.io/service-based-antipatterns/

and services hence the incoming as well as the outgoing coupling is high for bottleneck services. In addition, due to large number of external clients the response time is also high. Consequently, due to huge traffic the availability of these services is also low[6]. We propose detecting it by mapping the number of different contexts in which the same service is requested.
- Bloated Service - A service having one large interface which consist of many datatype parameters. All of them perform different operation with cohesion [11]. The output for this type of services are having less reusability, testability and maintainability in other business processes[7]. In order to detect it, the algorithm proposed here should check services definition and set threshold for the number of types and parameters.
- Cyclic Dependency - A never ending cycle exist between the services. [14].
- Service Chain - It is also known as message chain. A chain of services that are reserved for one common functionality. When a client want to achieve his goal by requesting a number of consecutive services this chain appears[8]. [8, 6]
- Shared Persistency - One database is shared for multiple services. [14]
- Stovepipe Service - Some of the functionalities are duplicated in on several services [11]. In this type of antipatterns a large number of protected and private methods are available that perform utility, infrastructure and business processes instead of focusing on the main goal [9].
- The Knot - This type of antipattern consist of a set of low cohesive services. All of these low cohesive services are tightly coupled due to which its reusability is very limited. These type of anti patterns have a complex infrastructure due to which its availability is low have high response time[10]. [12, 8, 11]

3 General Structure

Apart of detecting anti-patterns, we propose a methodology for retrieving the general structure of the project in terms of resources. A good solution should distribute resources along different files (library imports, method calls, etc.). In order to achieve this, we have modeled the project as a complex network, in which resources are represented by nodes and the paths by their usage.

A general description of the network is given by two metrics, Closeness and Betweenness:

Closeness Centrality (C) of a node in the network is the reciprocal of the sum of the shortest path distances relative to all other nodes.

[6] http://sofa.uqam.ca/resources/antipatterns.php#Bottleneck%20Service
[7] http://sofa.uqam.ca/resources/antipatterns.php#Bloated%20Service
[8] http://sofa.uqam.ca/resources/antipatterns.php#Service%20Chain
[9] http://sofa.uqam.ca/resources/antipatterns.php
[10] http://sofa.uqam.ca/resources/antipatterns.php

$$C(u) = \frac{n-1}{\sum_{v=1}^{n-1} d(v,u)} \quad (1)$$

The $d(v,u)$ is the shortest path between v and u and n is the number of nodes in the graph. High values for closeness represent central nodes in the network.

Betweenness Centrality (B) is the measure of the centrality of a node in a network, and can be calculated as the proportion of shortest path between node pairs that pass thought one node of interest.

$$C_B(v) = \sum_{s \neq t \neq v \in V} \frac{\alpha_{st}(v)}{\alpha_{st}} \quad (2)$$

α_{st} is the number of shortest paths from s to t, and $\alpha_{st}(v)$ is the number of shortest paths from s to t that pass through node v. Nodes with high betweenness centrality are central in the network and attract the information flow.

4 Case Study

As a case study, we analyzed Spinnaker project source code[11] for detecting the selected anti-patterns and retrieve its general structure. We assume services implemented as classes and a handcrafted set of words as forbidden for being used as class or method names. The code is available for reproducing the experiment[12].

In a first round, the proposed algorithm iterates through every file inside the project folder for detecting classes and imports names (in this case it is adjusted for detecting only .py files). In a second round it applies a regular expression for detecting occurrence of classes and its methods along the project and store all occurrences in a dataframe. We differentiate between methods declared inside classes from the ones declared outside. This provide sufficient information for evaluating services independently.

We have detected 3 hard-coded IP addresses and 82 version numbers. The project imports 86 different libraries, as showed in Figure 2 . We have detected three methods named as *delete*, *now* and *run*, considered as inappropriate.

We set the maximun size of a class name as 30, as well as for methods, for considering it too long. We have detected 3 methods and 13 classes with too longs names. An example of a method name is *determine_subprocess_outcome_labels*, and of class name *GcsArtifactStorageConfiguratorHelper*. We argue that a name with 30 characters long is not exactly big, but was selected as a threshold here as an example.

We count the number of parameters for each method for detecting too big interfaces (it was not possible to detect how many types in the case of python language). We set 5 arguments as the maximun allowed, and detected only one method, names *start_subprocess*.

[11] https://github.com/spinnaker/spinnaker
[12] https://github.com/rcaborges/microservices-antipatterns

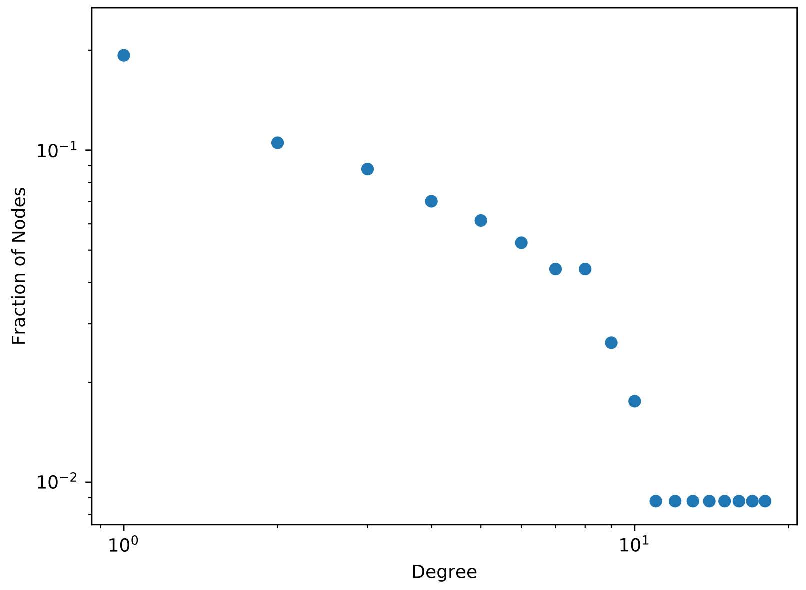

Fig. 1. Classes declared inside the project were modeled as a complex network. Fraction of nodes associated to each degree value.

As the last anti-pattern, we detect methods and classes instantiated from too many files. We set a threshold of 30 as a big number of files, and found 35 files associated to a method called *init_argument _parser*, 61 files associated to a class named *ConfigError*, and 31 files associated to *UnexpectedError*.

Regarding the general structure of the project in terms of resources, we build a network out of the information acquired in the previous phase. Each node represent one specific resource, ex. classes, and the connection between nodes are formed when two files use the same resource, ex. instantiate the same class.

Figure 1 show with logarithmic scale the fraction of nodes for each value of degree for the case of classes declared inside the project. Few classes were found in too many files, and most of them were detected in one single file. The shape of the curve indicates regular distribution along the files in the project.

Another information about the general structure of the project that may help project managers are the closeness and betweennes of the resources in the project. High values of closeness indicate resources that are close to others, and high values of betweenness represent resources used by too many files, located in the center of the network.

ConfigError was detected as having highest closeness and betweenness, and seems to be a widely used class along the project. Resources with high values of betweenness represent high risk, once it is central to the project and its failure

Table 1. Top 10 Closeness Indexes.

Class	**Value**
ConfigError	0.526611
UnexpectedError	0.359513
BranchSourceCodeManager	0.356089
HalRunner	0.333834
GcsPubsubNotficationConfigurator	0.325125
KubernetesConfigurator	0.325125
AwsConfigurator	0.325125
ArtifactConfigurator	0.325125
HaConfigurator	0.325125
GooglePubsubConfigurator	0.325125
AppengineConfigurator	0.325125

Table 2. Top 10 Betweenness Indexes.

Class	**Value**
ConfigError	0.206568
BranchSourceCodeManager	0.028653
UnexpectedError	0.023537
GitRepositorySpec	0.013929
HalRunner	0.006398
TimeoutError	0.003358
ExecutionError	0.002373
SemanticVersion	0.002226
BomBuilder	0.001973
RepositorySummary	0.001191

would impact great part of the system. It is worth mentioning that most of top 10 closeness indexes are configuration classes that need to operate in many different contexts inside the project.

Figure 2 shows the visualization of both networks, imports and classes, as described earlier. It is possible to notice that libraries are imported from quite many files inside the source code. Classes, on the other hand, show a more sparse usage among the files, as expected for a good microservice architecture.

5 Conclusions

An algorithm was presented for detecting anti-patterns in microservices based solutions. We have selected 5 anti-patterns and tried to detect them automatically in order to avoid potential failure points when migrating from monolithic architectures. We have also proposed a framework for describing the general structure of the project in terms of resource consumption, which should help project managers with a general overview.

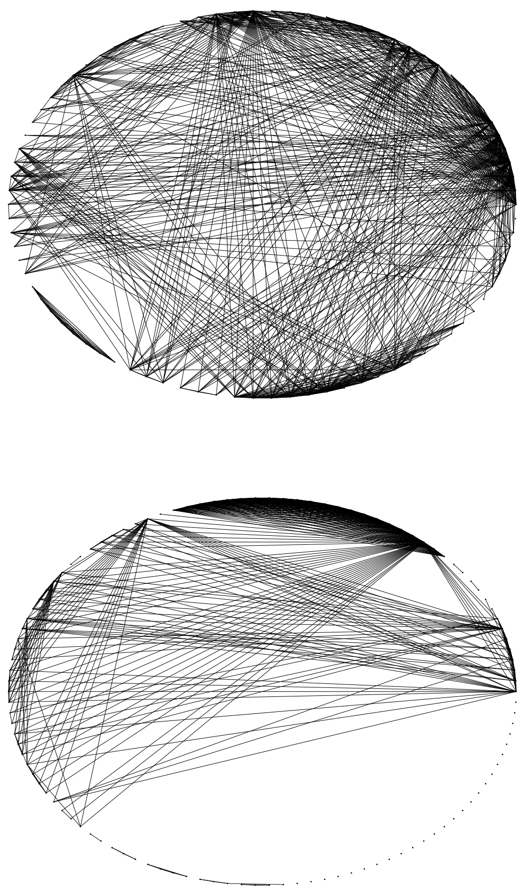

Fig. 2. Top: A visualization of how imports are distributed in different files along the project. Each node is one imported library and each connection represents two files importing the same library. Bottom: same visualizations but for classes declared inside the project code. Each node is one class (interpreted as a service), and each line corresponds to two files calling the same class.

We still need to consider the situations where classes are imported with different names, in this case there should be a proxy for considering extra calls for the original class.

We have simply detected IP addresses and version numbers, without providing any further analysis of the context these are happening. As a future work we plan to improve the algorithm so it can differentiate, for example, a request from a declaration of an IP address, or keep track of version numbers for suggesting consistency.

References

1. Arsanjani, A.: Service-oriented modeling and architecture. IBM developer works **1**, 15 (2004)
2. Bennett, K.H., Rajlich, V.T.: Software maintenance and evolution: a roadmap. In: Proceedings of the Conference on the Future of Software Engineering. pp. 73–87. ACM (2000)
3. De Win, B., Piessens, F., Joosen, W., Verhanneman, T.: On the importance of the separation-of-concerns principle in secure software engineering. In: Workshop on the Application of Engineering Principles to System Security Design. pp. 1–10 (2002)
4. Girard, J.F., Koschke, R.: Finding components in a hierarchy of modules: a step towards architectural understanding. In: 1997 Proceedings International Conference on Software Maintenance. pp. 58–65. IEEE (1997)
5. Jaspan, C., Jorde, M., Knight, A., Sadowski, C., Smith, E.K., Winter, C., Murphy-Hill, E.: Advantages and disadvantages of a monolithic repository: a case study at google. In: Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice. pp. 225–234. ACM (2018)
6. Jones, S.: SOA anti-patterns (2006), https://www.infoq.com/articles/SOA-anti-patterns
7. Long, J.: Software reuse antipatterns–revisited. Software Quality Professional **19**(4) (2017)
8. Nayrolles, M., Moha, N., Valtchev, P.: Improving soa antipatterns detection in service based systems by mining execution traces. pp. 321–330 (2013). https://doi.org/10.1109/WCRE.2013.6671307
9. Ouni, A., Kessentini, M., Inoue, K., Cinnide, M..: Search-based web service antipatterns detection. IEEE Transactions on Services Computing **10**(4), 603–617 (7 2017). https://doi.org/10.1109/TSC.2015.2502595
10. Palma, F., Moha, N., Tremblay, G., Guéhéneuc, Y.G.: Specification and detection of soa antipatterns in web services. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) **8627**, 58–73 (2014)
11. Palma, F., Mohay, N.: A study on the taxonomy of service antipatterns. pp. 5–8 (2015). https://doi.org/10.1109/PPAP.2015.7076848
12. Rotem-Gal-Oz, A., Bruno, E., Dahan, U.: SOA patterns. Manning (2012)
13. Sharma, T., Samarthyam, G., Suryanarayana, G.: Applying design principles in practice. In: Proceedings of the 8th India Software Engineering Conference. pp. 200–201. ACM (2015)
14. Taibi, D., Lenarduzzi, V.: On the definition of microservice bad smells. IEEE software **35**(3), 56–62 (2018). https://doi.org/10.1109/ms.2018.2141031
15. Taibi, D., Lenarduzzi, V., Pahl, C.: Architectural patterns for microservices: A systematic mapping study. In: CLOSER (2018)
16. Taibi, D., Lenarduzzi, V., Pahl, C.: Continuous architecting with microservices and devops: A systematic mapping study. In: Muñoz, V.M., Ferguson, D., Helfert, M., Pahl, C. (eds.) Cloud Computing and Services Science. pp. 126–151. Springer International Publishing, Cham (2019)
17. Wieringa, R.: Traceability and modularity in software design. In: Proceedings Ninth International Workshop on Software Specification and Design. pp. 87–95. IEEE (1998)