Policy Enforcement upon Software Based on Microservice Architecture

Tugrul Asik
Computer Engineering Department
Yildiz Technical University
Istanbul, Turkey
Email: tugrulasik@gmail.com

Yunus Emre Selcuk
Computer Engineering Department
Yildiz Technical University
Istanbul, Turkey
Email: yselcuk@yildiz.edu.tr

***Abstract*—Microservice is an architectural style that has recently started gaining popularity to become a new architectural phenomenon. Microservice architecture provides new opportunities to deploy scalable, language free and dynamically adjustable applications. This type of applications consist of hundreds or more of service instances. So that, management, monitoring, refactoring and testing of applications are more complex than monolithic applications. Therefore, some metrics and policies for measuring the quality of an application which is based on microservice architecture is needed. Moreover, automated tools are needed to carry out those tasks and enforce those policies. This work represents such metrics and policies. Additionally, an automated tool is implemented for automatic analysis of those metrics and policies upon software.**

***Index Terms*—Microservice architecture, software metrics, software policies, software quality, static analysis tool**

I. Introduction

In this study, microservice architecture was chosen to work on in order to measure software quality with newly defined metrics and policies. A Microservice based architecture is defined as a *software architecture pattern* for development of distributed applications, where the application is comprised of a number of smaller *independent* components; these components are small application in-themselves [1]. This architecture style has gained popularity over the last few years to describe a particular way of designing software applications as suites of independently deployable services [2]. There are certain common characteristics around organization around business capability, automated deployment, built released with automated processes, intelligence in the endpoints, and decentralized control of languages and data [2][3]. On the other hand, there is no precise definition of this architectural style and mentioned benefits come with challenges, such as discovering services over the network, security management, communication optimization, data sharing and performance [4]. Furthermore, team size, culture, skills, productivity etc. have an impact on software development [5]. Unfortunately, there is no strict rule about these challenges, there are some practices that have been proven in real-world systems. According to this characteristic features and trade offs, we studied on defining new metrics and policies to measure software quality and suitability for microservice practices. Metrics were defined to measure service size and inter-service communication.

In order to check compliance with these metrics and policies, we have implemented static analysis tool to analyze software based on microservice architecture. Finally, we have developed a sample ticket selling application to evaluate these metrics and policies, as well as to test our tool for functionality, not efficiency or analysis speed.

This paper is organized as follows. Section II describes the related works. Section III describes the details of defined metrics and policies. Section IV introduces our static analyzer tool. Section V describes the details about ticket selling application and its architecture. Section VI combines the results about experimental study. Finally, Section VII presents our conclusions.

II. Related Works

There are many metrics and policies to detect code anomalies, antipatterns and architectural problems for software products. In ISO9126-1 specification which represents the latest (and ongoing) research into characterizing software for the purposes of software quality control, software quality assurance and software process improvement (SPI) 6 main quality characteristics are identified. These are *functionality, reliability, usability, efficiency, maintainability, portability*. In object oriented programming; coupling, cohesion, encapsulation, inheritance and complexity are some of the popular metrics [6].

Another work relates architecture technical debt (i.e., result of architectural smells) with modularity metrics in order to estimate quality attributes [7] such as evolvability and maintainability. Moreover, especially in papers about microservices concerning scalability, reusability, maintainability, manageability and deployment quality attributes also used component/container, class and deployment UML diagrams to demonstrate the potential of implementing those attributes. Instance graphs/type graphs enabled to trace and validate quality attributes of health management, manageability and deployment automation. Dependency graphs co-occured with independence and maintainability quality attributes and also used to trace and test them [8].

In industry, real time monitoring metrics on a running system are popular, also important. Teams benefit from tools [9] to measure performance among service calls, to detect

978-1-5090-5756-6/17/$31.00 ©2017 IEEE
SERA 2017, June 7-9, 2017, London, UK

service failures and to track end to end business logic flow. Prometheus, AWS Cloud Watch, Metrics are some of the tools for these purposes. [10].

III. Metrics and Policies

Software metrics is defined by measuring of some property of a portion of software or its specifications. Software metrics provide quantitative methods for assessing the software quality [11]. Metrics are very useful for optimizing the performance of the software, debugging, managing resources, quality assurance, furthermore they leads to higher organizational performance [12] and visible results (such as those provided by measurements) are considered critical to success of any improvement plan, keeping participants [13] focused and motivated.

There is no strict policies in microservice architecture for service size, service call hierarchy, service discovery and security about software quality. Requirements, business flow changes, technology stack, team size [5], culture etc. have an impact on software.

In our study, there are three category for measuring and tracing the software quality in order to build suitable products for microservice architecture. First one is about measurement of service size, second one is about inter-service communications and the last one is about bad practices.

A. Measurement of Service Size

There are many methods to measure size of a software such as Source Lines Of Code (SLOC), Logical Lines Of Code (LLOC), COSMIC, Use Case-based etc. [14][15]. In order to measure a microservice size, our approach is that we count resources and clients which responsible for interactions between microservices or external services.

1) Resource Count (RC): Resource is defined as that a network data object or service that can be identified by a URI [16]. It represents a method which handles received requests to a service identified by a URI. Results will be the same for each service which is coded with any other programming language for this metric. Also it gives us the ability of tracking the service size changes or it can be an indicator for the need of splitting service into two or more smaller services.

2) Client Count (CC): Client is defined as that a program that establishes connections for the purpose of sending requests [16]. Client Count represents counts of HTTP service calls identified by a URI. Results will be the same for each service which is coded with any other programming language for this metric. Also, similar to the resource count criteria, the client count criteria gives us the ability of tracking the service resource changes and it can be an indicator for the need of splitting service into two or more smaller services. Furthermore, it's important for a service to know that how many services it needs in order to work properly.

Size (S) of a microservice is sum of these two metrics defined as *Resource Count (RC)* and *Client Count (CC)*, $S = RC + CC$. To describe size of a service in more detail, interactions of Sample Service is shown in Fig.1.

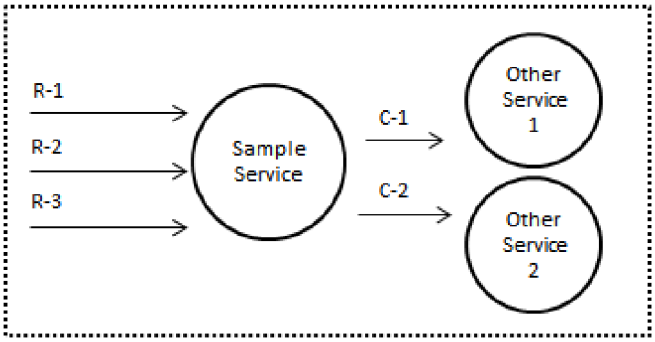

Fig. 1. Snippet-1 for service interactions of Sample Service

The service has 3 resources to handle received requests. These are *"GET sampleservice/samples"*, *"POST sampleservice/samples"*, *"GET sampleservice/samples/id/:id"*, R-1, R-2 and R-3 respectively. Moreover, Sample Service has 2 clients to send requests to other microservices that are mentioned as Other Service1 and Other Service2. Clients are *"GET otherservice1/others"* and *"GET otherservice2/others/id/:id"*. According to prementioned *service size* definition, size of the Sample Service is 5 (5 = 3 + 2).

B. Inter-Service Communication Compatibility

Microservices send messages to each other over HTTP[2]. They need to know each other and they have to send messages to each other in acceptable format with valid (existing) URIs in practice. Our approach in order to analyze compatibility is counting the unused resources and unreachable endpoints. Virtually prepared inter-service connection dependency graph which shows service interactions between services can be used to collect results for defined metrics. In that graph, URIs, transferred entities [16] and media types are used for mutual relation matching criteria.

1) Unused Resource Count (URC): It represents that a URI which is implemented in a microservice to handle received HTTP requests that isn't used by any other microservice in application domain. In other words, there is an open door in the microservice and nobody goes through it, but somebody is able to. This is a type of unused functionality, also potential security risk. Additionally, ideal architectural design should have zero unused resource count.

2) Unreachable Endpoint Count (UEC): It represents that a URI which is defined in a microservice to send HTTP request to an other microservice that is not exist or not matching to any URI criteria in application domain. Service request will face a problem and it will return wrong response to requester. It may cause reliability and functionality problems.

To describe inter-service communication compatibility in more detail shortly, artificially implemented case is shown in Fig.2. Sample Service has one resource mentioned as *R-4* that it is not used anymore by any other service, but It is still in source code. So that, Unused Resource Count (URC) is 1 for Sample Service. Moreover, Sample Application has one client

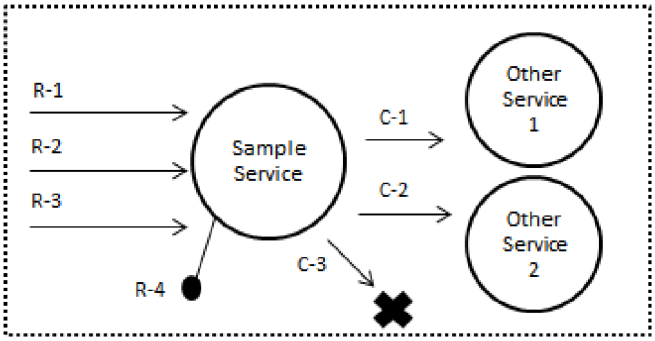

Fig. 2. Snippet-2 for service interactions of Sample Service

mentioned as *C-3* to send a request to an other service but URI (end point) is not correct or is not implemented yet by any service. It means that Unreachable Endpoint Count(UEC) is 1 for Sample Service. In short, URC is 1 and UEC is 1 for that sample scenario.

General purpose of these metrics in this part is that to improve maintainability and reliability of software. In other words, we want to avoid bad coding, refactoring and design practices. Moreover, our other purpose is to keep inter-service communication under control and make software less affected by continuous source code changes.

C. Bad Practices

The following practices are discouraged in microservice implementations. Interfaces provided by microservices should be *discoverable*, consumer of the interfaces must be able to look up, find the interfaces without having explicitly knowledge of the underlying technology implementation or location [16].

1) Static URI (SURI): This practice represents the client URI to send messages to an other microservice that is defined in a the service statically. In other words, service URI of an other service is hard coded into source code. For horizontal scaling, services live up dynamically. So that, service URI changes dynamically, too. Because of these reasons, any microservice should not has static IP address of an other service. It may causes communication problems.(e.g static IP *http://10.10.11.12:8080/sampleservice/sample*)

2) Long URI (LURI): Technically, according to RFC-2616 [16], the HTTP protocol does not place any a priori limit on the length of a URI. Services should handle received requests by meaningful and short URIs. In microservice architecture, URIs are important for service discovery and usability, also connectivity. It should not be too long and it should be easy discoverable, meaningful and clear.

Policies. According to these metrics in this section, a set of rule was defined. Service size should be less than 15, Unused Resource Count(URC) and Unreachable Endpoint Count(UEC) should be 0. Services should not have any static IP addresses of other services. Service URIs to provide functionalities should not be more than 50 characters including dynamic parameter keywords.

IV. Static Code Analyzer Tool

Program analysis technology has been proposed to detect bugs in software. Based on whether the target program will be running, program analysis can be divided into dynamic program analysis and static analysis [17]. Static code analysis method was chosen to analyze specified metrics and a static analyzer tool was developed in this direction. Static analysis tools provide a means for analyzing code without having to run the code, helping ensure higher quality software throughout the development process. There are a variety of ways to perform automatic static analyses, including at the developers request, continuously while creating the software in a development environment [18]. There are many approaches for analyzing software to find bugs, bad smells and to measure software quality.

The tool takes source code written in Java as input, analyzes it and displays the results for specified metrics and policies. Aforementioned Java input was developed with Dropwizard Framework. JavaParser library was used for parsing the code to get Abstract Syntax Tree (AST). AST is reduced, and then every reduced part is assigned to an object which represents attributes for aforementioned metrics. The reduced information includes HTTP method, Java method name, line location in Java file etc., including raw AST in case of customized information may be needed. List of microservices is defined in a configuration file which identified by the tool. The tool analyses all defined microservices together to list metric results.

V. Sample Application Setup

Ticket Selling application for cinema and theatre is implemented as microservices with some bugs and bad practices so that it can be analyzed by our static analyzer tool. There are eight services in the sample application. On the other hand, real world applications have usually more services depending on the project requirements. Our services are API gateway, salon, ticket, credit card, debit card, coupon, SMS and mail service. Also, we have two external service providers named as Banks and Sender Provider. These are for payment, SMS and mail operations respectively. They are out of scope for static analysis. High level application architecture is given in Fig. 3. Moreover, services are independent and they send messages to each other over HTTP on demand.

We assumed that Domain names (DNS) are dynamic and discoverable by each service in the application domain excepts bad practices scenarios. Our RESTful API for messaging is suitable for Richardson Maturity Level-II which uses the HTTP verbs[19] [20]. Furthermore, data layers of services are designed suitable for microservice architecture, decentralized.

Information about *microservices* and *labeled policy checks* like *Unused Resource* are given in service details. RESTful API paths(URIs) are formatted as declarative with the format ”HTTP Method + WhiteSpace + Service URI” e.g *GET sampleservice/samples*. Double dot (:) is used for prefix for dynamic parameters e.g *:id*. Resource and client lists are defined in details of each microservice.

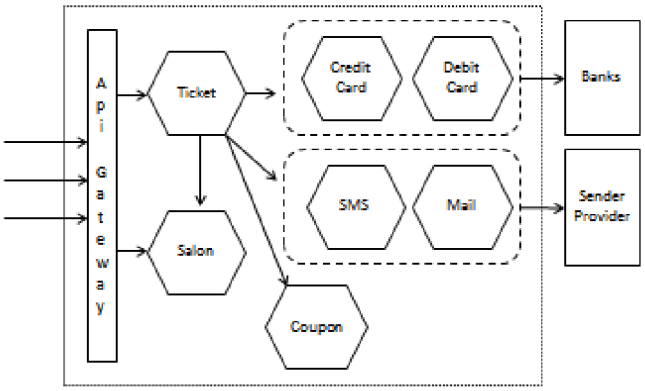

Fig. 3. High level microservice architecture for service interactions

API Gateway

It's a service designed for a frontier handler of all requests from clients. Services aren't called directly from external clients. Some services are in private side. So that, we assumed that API gateway includes all endpoints of these two services, Salon and Ticket services.

Salon Service

It's the service designed for providing information about salons and shows.

Resources are;

- GET /salonservice/salons
- GET /salonservice/salons/:id
- GET /salonservice/salons/cities/:cid
- PUT /salonservice/salons/:id/seats/:sid
- GET /salonservice/salons/cities/:cid:/district/:did /category/:caid/time/from/:fromDate/to/:toDate/... (*Long URI*)

Ticket Service

It's the service designed for creating tickets.

Resources are;

- GET /ticketservice/tickets
- GET /ticketservice/tickets/:id
- GET /ticketservice/tickets/seats/:id
- POST /ticketservice/tickets
- PUT /ticketservice/tickets
- DELETE /ticketservice/tickets/:id

Clients are;

- PUT /salonservice/salons/:id/seats/:sid
- POST /creditcardservice/cards
- POST /debitcardservice/cards
- PUT /couponservice/coupons
- POST /smsservice/smses
- POST /pushnotificationservice/notification (Unreachable)
- POST http://10.11.12.13/mailservice/mails (*static URI*)

Credit Card Service

It's the service that accepts payment with credit card. It doesn't have its own database. Assumed that it is connected with the banking APIs.

Resources are;

- POST /creditcardservice/cards
- POST /creditcardservice/visa (*unused resource*)
- POST /creditcardservice/master (*unused resource*)
- POST /creditcardservice/americanexpress (*unused resource*)

Debit Card Service

It's the service that accepts payment with debit card. It doesn't have its own database. Assumed that it is connected with the banking APIs.

Resources are;

- POST /debitcardservice/cards

Coupon Service

It's the service that accepts payment by coupon code.

Resources are;

- PUT /couponservice/coupons
- GET /cooponservice/coupons (*unused resource*)

SMS Service

It's the service to send short messages.

Resources are;

- POST /smsservice/smses

Mail Service

It's the service to send e-mails.

Resources are;

- POST /mailservice/mails

VI. Evaluating The Results

Bad practices and detectable cases are created on purpose in the Ticket Selling application. Analysis results are shown in Table I, these results were listed by our analyzer tool.

Ticket Service has 6 resources, Salon Service has 5, Credit Card Service has 4, Coupon Service has 2 and others have just 1 resource(s). Furthermore, Ticket Service has 7 clients and others have 0 client excepts external interactions. From this viewpoint, Ticket Service needs other services to work properly.

We see that Ticket Service is the biggest service by making use of the referred definition of microservice size. It is suitable for our policy(size should be less than 15) about microservice size. On the other hand, results do not show that Ticket Service has more source lines of code than other services or Mail Service has less logical lines of code than Salon Service.

TABLE I
METRIC BASED ANALYSIS RESULTS OF TICKET SELLING APPLICATION

Name	RC	CC	Size	URC	UEC	SURI	LURI
Salon	5	0	5	0	0	0	1
Ticket	6	7	13	0	1	1	0
Credit Card	4	0	4	3	0	0	0
Debit Card	1	0	1	0	0	0	0
Coupon	2	0	2	1	0	0	0
SMS	1	0	1	0	0	0	0
Mail	1	0	1	0	0	0	0

Our measurement of size definition is not dependent with other size measurement methods.

In Credit Card and Coupon services, there are resources which are not used by any other microservice. That case breaks our rules, services should have 0 unused resource. There are many possible scenarios for that case. One of the these scenarios is that developer may forget to remove or refactoring the unused functionalities, other one is that code bases of these services may be released earlier than others.

Ticket Service has a client but it does not exist in the application domain. It includes a client to send notifications with the request *POST /pushnotificationservice/notifications*. That case may cause functionality problems. Possible scenarios are that the URI may not be written correctly or push notification service may be removed from service list. But, Ticket Service still has the unreachable client into the source code. Even this client does not affect anything bad, it needs to be removed in order to make source code clean and readable.

Salon Service has a resource to handle complex queries, but its URI violate our URI length policy. It is longer than 50 characters. It should be shorter if it is possible. if not, it can be marked as an exception.

Ticket Service has a client to send request to Mail Service. Client URI is defined as hard coded. The URI is *http://10.11.12.13/mailservice/mails*. If Mail Service lives up in an other URI, Ticket Service will have a mail sending problem. It means that you may fail in production just because of a hard coded URI. Moreover, It's hard to detect quickly these type of hard coded configuration problems. So that, Static URI(SURI) metric which is categorized as bad practices may save the day.

We can generate many scenarios for policy violations mentioned above. It is important to understand that early detections and automated analyze with tuned policies is helpful to manage software quality.

VII. CONCLUSION

In this paper, according to the characteristic features of microservice architecture we discovered some gray areas. From this viewpoint, new metrics and policies were defined. These are *Resource Count (RC)*, *Client Count (CC)* are defined for measurement of service size; *Unused Resource Count (URC)*, *Unreachable Endpoint Count (UEC)* are defined for service interconnection compatibility; *Static URI (SURI)*, *Long URI (LURI)* are defined for bad practices. Sample application were implemented with some bugs and bad practices on purpose. In order to compliance with these metrics, software were analyzed by our static analysis tool. Experimental results and expectations were compared and evaluated. Results are as same as expected.

Although sample application is a small application, many problematic scenarios are generated and they are detected by the tool. There are more microservices than our sample application in real world. It is important to understand that automated software quality management is important to make software under control. Also running tuned policies upon a software makes it more visible and observable to prevent bugs and bad smells.

REFERENCES

[1] Dmitry Namiot, Manfred Sneps-Sneppe, "On Microservices Architecture", International Journal of Open Information Technologies ISSN: 2307-8162, vol. 2, no. 9, pp. 24-27, 2014.

[2] James Lewis, Martin Fowler, "Microservices", https://martinfowler.com/articles/microservices.html, 2014. [Online; accessed 16-Mar-2017].

[3] I. Nadareishvili et al., "Microservice Architecture: Aligning Principles, Practices and Culture", OReilly, 2016.

[4] Johannes Thnes, "Microservices", IEEE Software, vol. 32, no. 1, pp. 116-116, 2015.

[5] P. Clarke, R. V. O'Connor, "Changing Situational Contexts Present a Constant Challenge to Software Developers." in Systems Software and Services Process Improvement, Springer International Publishing, pp. 100-111, 2015.

[6] ISO 9126 Software Quality Characteristics, "An overview of the ISO 9126-1 software quality model definition, with an explanation of the major characteristics", http://www.sqa.net/iso9126.html. [Online; accessed 16-Mar-2017].

[7] Z. Li, P.Liang, P. Avgeriou, N. Guelfi, and A. Ampatzoglou. "An empirical investigation of modularity metrics for indicating architectural technical debt". In 10th International ACM SIGSOFT Conference on Quality of Software Architectures, QoSA14, 2014.

[8] Nuha Alshuqayran, Nour Ali, Roger Evans, A Systematic Mapping Study in Microservice Architecture, ISBN: 978-1-5090-4781-9, 4-6 Nov. 2016.

[9] Chris Richardson, *Application Metrics*, http://microservices.io/patterns/observability/application-metrics.html. [Online; accessed 16-Mar-2017].

[10] "AWS Cloud Watch", https://aws.amazon.com/cloudwatch. [Online; accessed 16-Mar-2017].

[11] J. Verner and G. Tate, A software size model, IEEE Transaction on Software Engineering, vol. 18, no. 4, 1992.

[12] Gopal, A., Krishnan, M., Mukhopadhyay, T., Goldenson, D.R. Measurement programs in software development: determinants of Success, IEEE Trans. Softw. Eng., 2002, 28, pp. 863 875.

[13] Iversen, J., Ngwenyama, O. , Problems in measuring effectiveness in software process improvement: a longitudinal study of organizational change at Danske data, Int. J. Inf. Manage., 26, (1), pp. 30 43, 2006.

[14] Charles Symons, Alain Abran, Christof Ebert, Frank Vogelezang, Measurement of Software Size: Advances Made by the COSMIC Community, IWSM-MENSURA, 2016.

[15] Software sizing, https://en.wikipedia.org/wiki/Software_sizing. [Online; accessed 16-Mar-2017].

[16] RFC2616 Specification, "Hypertext Transfer Protocol HTTP/1.1", https://tools.ietf.org/html/rfc2616. [Online; accessed 16-Mar-2017].

[17] Hongliang Liang, Lei Wang, Dongyang Wu, Jiuyun Xu, MLSA: a static bugs analysis tool based on LLVM IR, International Journal of Networked and Distributed Computing, Vol. 4, No. 3, pp. 137-144, Jul. 2016.

[18] M. Gegick and L. Williams, "Towards the use of automated static analysis alerts for early identification of vulnerability-and attack-prone components", in Proc. ICIMP, 2007, pp. 18-23.

[19] "Richardson Maturity Model", http://restcookbook.com/Miscellaneous/richardsonmaturitymodel/. [Online; accessed 16-Mar-2017].

[20] "Richardson Maturity Model Level-2", https://martinfowler.com/articles/richardsonMaturityModel.html. [Online; accessed 16-Mar-2017].