Identifying Availability Tactics To Support Security Architectural Designs in Microservices-based Systems

Conference Paper · July 2019

DOI: 10.1145/3344948.3344996

Gastón Márquez
Universidad Técnica Federico Santa María

Hernán Astudillo
Universidad Técnica Federico Santa María

Identifying Availability Tactics to Support Security Architectural Design of Microservice-based Systems

Gastón Márquez and Hernán Astudillo
Universidad Técnica Federico Santa María
Valparaíso, Chile
gaston.marquez@sansano.usm.cl,hernan@inf.utfsm.cl

ABSTRACT

Microservices is an architectural style that considers systems as modular, customer-centric, independent, and scalable suite of services. In order to address security requirements in microservices-based systems, architects often must focus on critical quality attributes, such as availability, aiming at employing architectural solutions that provide design decisions that address key security concerns (also known as *architectural tactics*). Although current architectural tactics for availability offer an extensive catalog of alternatives to improve availability and security factors, new availability concerns (emerging from security microservices requirements) demand new or improved architectural tactics. In this article, we examined the source code and documentation of 17 open source microservices-based systems, identified 5 uses of availability tactics, and characterized them using a newly introduced descriptive template. We found that almost all (4 out of 5) tactics did focus on preventing faults rather than detecting, mitigating or recovering from them (which are the traditional tactics taxonomies' branches). This approach can be further used to systematically identify and characterize architectural tactics in existing microservices-based systems in other critical quality attributes concerning security, such as confidentiality and integrity.

CCS CONCEPTS

• **Software and its engineering → Software architectures**;

KEYWORDS

Microservices, availability, architectural tactics, frameworks, patterns

ACM Reference Format:
Gastón Márquez and Hernán Astudillo. 2019. Identifying Availability Tactics to Support Security Architectural Design of Microservice-based Systems. In *European Conference on Software Architecture (ECSA), September 9–13, 2019, Paris, France.* , 7 pages. https://doi.org/10.1145/3344948.3344996

1 INTRODUCTION

The *microservices* architectural style is an approach to develop single applications as suites of small services, each running in its own process and communicating with lightweight mechanisms [13]. The complexity of microservices-based systems is determined partly by its functionality and partly by the global requirements on its development [20], such as security. Recent industrial reports[1] reveal that companies, such as Netflix, received massive attacks on their microservices-based systems in the last years. These attacks cover from specific services to the whole architecture. For this reason, companies are constantly creating new security mechanisms intending to protect themselves from such attacks.

According to [28], in order to satisfy the general security objectives, the following quality attributes must be addressed: integrity, confidentiality, and availability. Correspondingly, to build secure microservices architectures, architects should evaluate several decisions related to the aforementioned quality attributes, especially for availability [10].

In order to map availability properties into microservices architectures, architectural tactics emerge as an alternative. These are design decisions that influence the achievement of a quality attribute response [4], allowing the evolution of the software architecture or removing obsolete decisions to satisfy changing requirements. Architectural tactics have been used to provide decisions in big data cyber-security analytics systems [27], understanding software vulnerabilities [24], and the like.

The current availability architectural tactics catalog [4] provides several tactics to detect, prevent, and recover from faults when systems no longer deliver services that are consistent with its specifications. Nevertheless, specific availability requirements that support security strategies to handle security concerns in microservices-based systems require to update or propose architectural availability tactics.

In this article, we investigated which architectural availability tactics for microservices-based systems (from now, *microservices availability tactics*) emerge to support architectural design decisions related to security concerns in microservices architectures. We described a template to characterize and illustrate microservices availability tactics. In order to investigate the usage of these microservices availability tactics and their implications concerning security, we conducted an empirical study in 17 open microservices-based systems (OMbS). The main **contributions** of this article are (i) a template for characterizing and describing architectural tactics and (ii) 5 microservices availability tactics.

The remainder of the article is structured as follow: Section 2 describes background; Section 3 details the empirical study design; Section 4 illustrates results and discusses key findings; Section 5 describes threats to the validity; Section 6 shows related work; Finally, Section 7 concludes and discusses key findings.

ECSA, September 9–13, 2019, Paris, France
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-7142-1/19/09…$15.00
https://doi.org/10.1145/3344948.3344996

[1] https://www.whitehatsec.com/

2 AVAILABILITY TACTICS

In this section, we describe the background of our study.

2.1 Architectural tactics

Initial SEI's technical reports [3] [5] define architectural tactics "as a means of satisfying quality-response measure by manipulating some aspect of a quality model trough architectural design decisions". An example of an architectural tactic for availability [4] is:

Example 2.1. Heartbeat. *One component emits a heartbeat message periodically, and another component listens for it. If the heartbeat fails, the originating component is assumed to have failed, and a fault correction component is notified.*

The procedure on how architectural tactics were created is not explicitly described in the literature. Technical reports explain that architectural tactics were obtained from interviews with experts and architects and their usage was reported in several case studies. Nevertheless, to the best of our knowledge, there is not a standard structure to define them.

After reviewed several technical reports and positions papers (such as [2]) between the years 2000 and 2005, we realized that architectural tactics are composed by the following properties:

- *Stimuli*: Something that incites to action or exertion or quickens action.
- *Environment*: The aggregate of surrounding things, conditions, or influences; surroundings; milieu.
- *Response*: An answer or reply, as in words or in some action

Following this structure, Example 2.1 can be rewritten as:

- *Stimuli*: The fault of normal operations between the system's components.
- *Environment*: System's normal operation.
- *Response*: One component emits a heartbeat message periodically, and another component listens for it. If the heartbeat fails, the originating component is assumed to have failed, and a fault correction component is notified.

2.2 Tactics, patterns, and frameworks

Patterns (such as design, architecture, and others) are systematic heuristics that provide recurring solutions to common problems [7]. From an architectural perspective, patterns are often used to achieve quality attributes and making more explicit the architectural decision making in a system [4]. Due to patterns represent reusable architectural and design knowledge, it seems reasonable to establish that they also represent pre-defined designs to be used in the development and/or implementation of an application [11]. In this context, application frameworks (or frameworks) often characterize these pre-defined designs. Frameworks are collections of reusable software elements that provide generic functionality addressing recurring domains and quality attributes concerns across a broad range of applications [12].

According to [4], there is an implicit relationship between patterns and tactics, *patterns are built from tactics*; therefore, patterns also provide the starting point for incorporation of tactics [9]. Moreover, in our previous work [17], we realized that some patterns are characterized by frameworks. Therefore, we assume that it is possible to discover architectural tactics in frameworks which implement specific patterns.

In order to obtain availability tactics for microservices-based systems, we started our study by (i) conducting literature reviews in academic as well as industrial sources to obtain patterns (from now, microservices patterns) [15] [16] and (ii) performing an empirical study in open microservices-based systems in order to identify frameworks that characterize microservices patterns [17] [14]. As a result, in (i) we obtained a set of 17 microservices patterns and in (ii), we recognized 23 frameworks which characterize one or more microservices patterns. In general, frameworks characterize one or more microservices patterns, but in this study, we focused on mentioning the patterns that are directly related to availability concerns.

2.3 Availability tactics in microservices-based systems

In our previous work [14], we identified 10 microservices patterns (related to 12 frameworks) that address availability concerns. Then, we proceed to read and examine their documentation focusing in to identify *design principles* implemented in the frameworks. We defined design principles to those design guidelines which satisfy the following attributes:

- *Stimuli, Enviroment, Response* (previously described in Section 2.1)
- *Architectural pattern*: The design principles should describe in which pattern(s) they are contextualized.
- *Quality attribute*: The documentation should illustrate which QA's the design guidelines help to achieve.
- *Parameters*: This field describes which variables are used to measure architectural tactics response.
- *Architectural elements*: This field complements the above one. It defines which elements are used to measure an architectural tactic response.

From the initial set of 10 microservices pattern, we found evidence of availability microservices tactics in 3. These microservices patterns and their corresponding frameworks are:

- *Circuit breaker*: The Circuit Breaker pattern checks the health status or remembers the number of unsuccessful calls and trips if a certain threshold is reached. If the circuit breaker is triggered, it will return an error instead of sending the call to the remote service, to prevent the broken service to be penetrated with additional requests. Framework: Netflix Hystrix[2].
- *Service Registry*: The Service Registry pattern maps, between a unique identifier and the current address of a service, instances in order to decouple the physical address of service from the identifier. Frameworks: Netflix Eureka[3] and Apache Turbine[4].

[2] https://github.com/Netflix/Hystrix
[3] https://github.com/Netflix/eureka
[4] https://turbine.apache.org

- *Messaging*: This pattern suggests using asynchronous messaging for inter-service communication. Message queue allows the state to be asynchronously and reliably sent to different locations. Frameworks: RabbitMQ[5] and Redis[6].

From the 3 aforementioned microservices patterns and their respective frameworks, we obtained 8 design principles that are interpretable as *tentative* microservices availability tactics. Aiming to improve and refine the microservices availability tactics, we interviewed 5 practitioners intending to know which decisions they make regarding availability issues and microservices availability tactics. For this, we analyzed 3 availability-based scenarios corresponding to projects performed by them. Then, we presented to them the tentative microservices availability tactics, and for each scenario, we evaluated the advantages and disadvantages of each tactic. After brainstorming sessions between the practitioners and our research team, we obtained 5 microservices availability tactics, which are described in Table 2.

Table 1: Profile and experience (years) of practitioners

	Domain	Work exp.	Microservices exp.
Practitioner 1	Retail sales	20+	6
Practitioner 2	Retail sales	10	6
Practitioner 3	Retail sales	3	2
Practitioner 4	Retail sales	3	2.5
Practitioner 5	Banking	8	3

We omitted "Quality attribute" in Table 2 because these tactics are related to the same quality attribute (availability). On the other hand, some tactics (such as *self-preservation* and *asynchronous messaging*) are variants from well-know tactics [4] (such as *Heartbeat*, *Ping/Echo*, *Monitor*, *Load balancing*). Finally, we categorized the microservices availability tactics using the same categories for availability tactics described in [4] (see Figure 1)

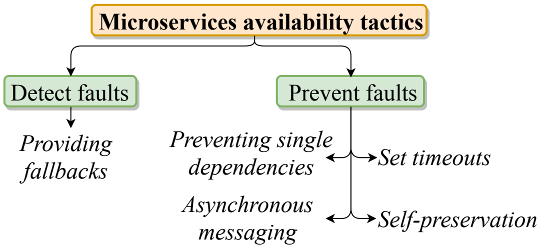


Figure 1: Microservices availability tactics catalog

3 STUDY DESIGN

This section describes the design of our empirical study.

3.1 Goal and research questions

The goal of this study is to *identify microservices availability tactics in open microservices-based systems*. Subsequently, the research questions are:

RQ1: *How many microservices availability tactics is possible to recognize in source code?* By answering this research question, we aim at illustrating the frequency of availability tactics usage in source codes corresponding to the projects set described in Table 3.

RQ2: *How many microservices availability tactics is possible to recognize in projects' documentation?* Following the same idea of RQ1, the goal of this research question is to describe the frequency of availability tactics identification in the projects' documentation. When we refer to *the documentation*, we include README files, projects' open and closed issues, wikis, and others.

3.2 Scope

The scope of this study is related to microservices projects in the context of Open Source Software (OSS). These kinds of projects have followed recent years of growth among users and companies that integrate open source software into their activities. Its multiple advantages and dynamism make OSS's an attractive work tool for all types of companies and developers [25].

The research period of this study began in November 2018 and ended in April 2019.

3.3 Procedure

3.3.1 Projects selection. Using the GitHub platform as a source of open projects, we obtained 17 open microservices-based systems (see Table 3). We used the following inclusion and exclusion criteria for each project:

- *Inclusion criteria*: Benchmark requirements for microservices projects proposed by Aderaldo et al. [1], and projects with source code available.
- *Exclusion criteria*: Projects with no robust information (basic examples, projects in process, others), tools to build microservices (instead of frameworks), component-based projects to build microservices (e.g., projects just for gateway components to microservices), and projects used as an example for lectures or talks.

3.3.2 Analysis. To conduct the analysis, (i) we reviewed the documentation and (ii) we examined the structure and source code (see Figure 2).

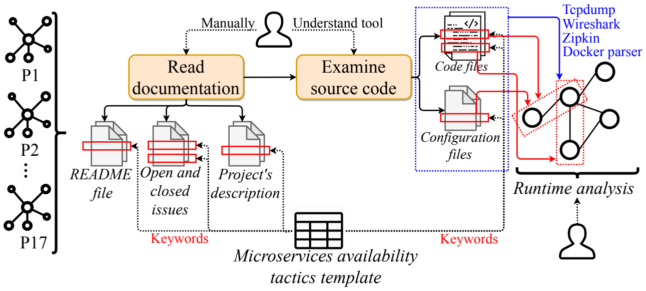


Figure 2: Analysis overview

[5] https://www.rabbitmq.com
[6] https://redis.io

Table 2: Microservices availability tactics

Id	Tactic	Stimulus	Environment	Response	Arch. pattern	Parameters	Arch. elements
T1	*Preventing single dependency*	Latency in transmitting or processing data in a network	Communication over a network	Prevent single dependencies by wrapping all calls to external systems (or dependencies) in an object which typically executes within a separate thread	Circuit breaker	Number of dependencies	Services, dependencies
T2	*Set timeouts*	Timing-out calls that crash services	Microservices communication	Set timing-out calls that take longer than thresholds arbitrarily defined by the architect	Circuit breaker	Number of timing out calls	Dependencies
T3	*Providing fallbacks*	A client service repeatedly suffers dependency faults	Microservices communication	Maintain a small thread-pool for each dependency. If it bemes full, requests destined for that dependency will be immediately rejected instead of queued up	Circuit breaker	Number of dependency failures	Dependencies
T4	*Self-preservation*	An unexpected number of registered service clients fail in their connections and are pending eviction at the same time	Catastrophic network events	Execute an explicit unregister action when service clients are permmanently going away. Any service client that fail 3 consecutive heartbeat renewals is considered to have an unclean termination	Service Registry	Servers operations failures	Services
T5	*Asynchronous messaging*	A service fails to respond to an client request	Microservices communication	Apply asynchronous messaging protocol. The client code or or message sender usually does not wait for a response	Messaging	Number of messages exchanged	Services

Table 3: Open microservices-based systems used in this study

ID	Projects	URL (https://github.com)
P1	Gizmo	/nytimes/gizmo
P2	Magda	/magda-io/magda
P3	Acme air	/acmeair/acmeair-nodejs
P4	Sock Shop	/microservices-demo/microservices-demo
P5	Piggy Metrics	/sqshq/PiggyMetrics
P6	Apolo	/ctripcorp/apollo
P7	Share bike	/JoeCao/qbike
P8	eShop	/dotnet-architecture/eShopOnContainers
P9	Warehouse microservice	/HieJulia/warehouse-microservice
P10	Microservices Reference	/mspnp/microservices-reference-implementation
P11	Vehicle tracking	/mohamed-abdo/vehicle-tracking-microservices
P12	EnterprisePlanner	/gfawcett22/EnterprisePlanner
P13	Freddy's bbq joint	/william-tran/freddys-bbq
P14	Photo uploader	/nginxinc/mra-ingenious
P15	WeText	/daxnet/we-text
P16	Pitstop	/EdwinVW/pitstop
P17	SiteWhere	/sitewhere/sitewhere

Regarding (i), we focused on detecting the design decision made by the developers or architects reported in the documentation as well as in the open and closed issues of each project. In (ii), we used the Understand[7] tool aiming at helping to have a complete visualization of the source code and dependencies of each project. We performed 4 steps for the source code analysis: (1) Generate the architectural dependency diagram of the project, allowing to identify clusters (as potential microservices), (2) Identify the files related to each cluster, (3) Code analysis of these files to find their frameworks, and (4) Traceability analysis to locate source code files and framework references in the complete source code.

Example 3.1. Figure 3 illustrates (partially) the four steps executed in the project P5. In step (1), we identified four clusters composed by the architectural dependencies of the project. Note that these clusters are "tentative" functional microservices and not infrastructure microservices. We define *functional microservices* as those services that represent business-critical purposes. *Infrastructure microservices* are those microservices who belongs to the virtualization environments that support a microservices-based system. In step (2), we checked each source file corresponding to each cluster. In this example, we examined the second cluster. In step (3) we selected a file called `GatewayApplication.java` in order to visualize their dependencies and packages. Subsequently, in step (4) we used the features of the Understand tool in order to review which other classes are linked with `GatewayApplication.java` (such as `StatisticsApplication.java`, `AuthApplication.java`, and others) in order to find code snippets related to microservices availability tactics.

In order to complement the source code analysis, we proceed to check how many microservices per projects are related to microservices availability tactics. To perform this task, we used as a reference the *tentative* microservices obtained from the source code analysis and we manually trace the dependency between microservices analyzing the network microservices traces (Tcpdump[8] and Wireshark[9]), log traces (Zipkin[10]), and Docker logs (if the OMbS use docker-compose[11]) along with the IP address, ImageID, ports, and end-points of each microservice.

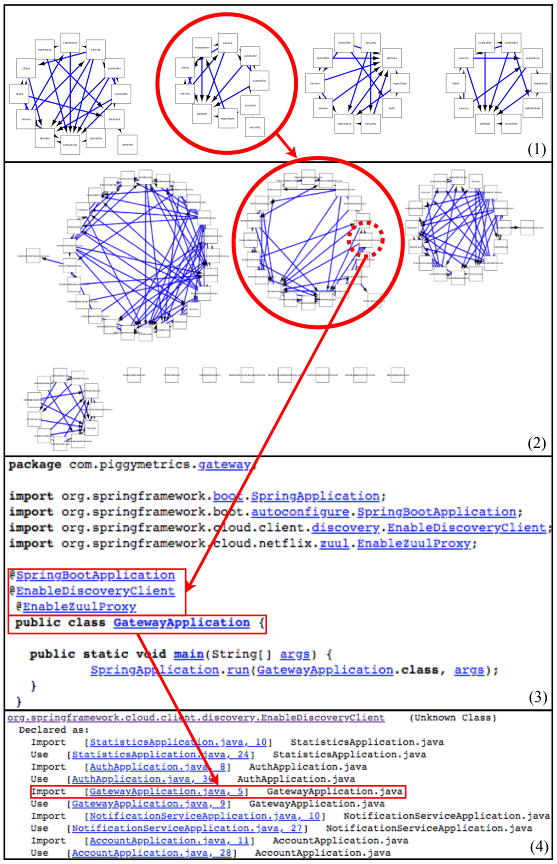
[7] https://scitools.com/features/

[8] https://www.tcpdump.org

[9] https://www.wireshark.org

[10] https://zipkin.apache.org

[11] https://docs.docker.com/compose/



```
package com.piggymetrics.gateway;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.client.discovery.EnableDiscoveryClient;
import org.springframework.cloud.netflix.zuul.EnableZuulProxy;

@SpringBootApplication
@EnableDiscoveryClient
 @EnableZuulProxy
 public class GatewayApplication {

   public static void main(String[] args) {
          SpringApplication.run(GatewayApplication.class, args);
   }
 }
```

```
org.springframework.cloud.client.discovery.EnableDiscoveryClient    (Unknown Class)
  Declared as:
    Import   [StatisticsApplication.java, 10]   StatisticsApplication.java
    Use   [StatisticsApplication.java, 24]   StatisticsApplication.java
    Import   [AuthApplication.java, 8]   AuthApplication.java
    Use   [AuthApplication.java, 39]   AuthApplication.java
    Import   [GatewayApplication.java, 5]   GatewayApplication.java
    Use   [GatewayApplication.java, 9]   GatewayApplication.java
    Import   [NotificationServiceApplication.java, 10]   NotificationServiceApplication.java
    Use   [NotificationServiceApplication.java, 27]   NotificationServiceApplication.java
    Import   [AccountApplication.java, 11]   AccountApplication.java
    Use   [AccountApplication.java, 28]   AccountApplication.java
```

Figure 3: Source code analysis' steps

4 RESULTS AND DISCUSSION

This section answers the research questions and discusses key findings.

4.1 RQ1: Microservices availability tactics and code

Figure 4 describes the percentage of microservices availability tactics' presence in the open projects as well as the programming languages which the evidence of microservices tactics is found, where the tactic T2 is the most used (65%), succeeded by T1 (47%), T5 (41%), T3 (35%), and T4 (12%).

The goal of this tactic is to provide mechanisms that allow stopping waiting for a response that will not come. At the time when we reviewed the source code, we realized that OMbS developers use this tactic because networks are fallible (projects P1, P5, and P17 emphasize this). Although T2 is the most used tactic, it is essential to set which will be the timeout value because confusions on this parameter may produce an *anti-pattern* [23].

On the other hand, the tactic with less presence is T4. The aim of this tactic to avoid poor network connectivity failure. Nevertheless, although it is reasonable to associate the usage of well-known architectural tactics to one or more application frameworks [12], this tactic is generally used by Netflix Eureka. The relationship between

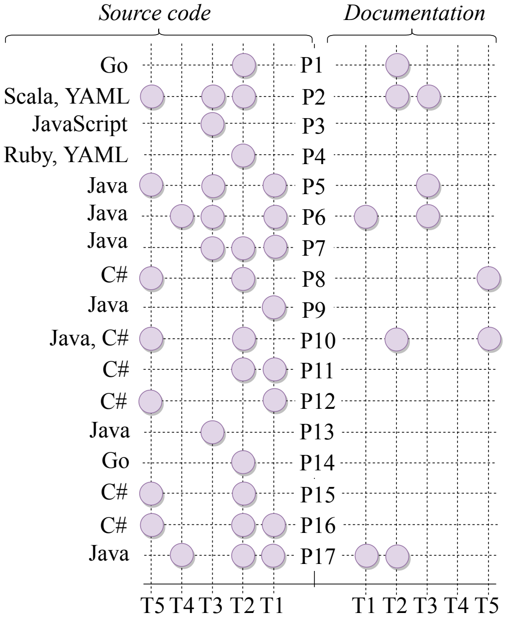


Figure 4: Presence of microservices availability tactics in source code and documentation

T4 and the aforementioned framework is that Eureka servers stop terminating instances from the service registry when they do not receive heartbeats beyond a certain threshold. Therefore, the usage of T4 depends on if the microservices-based system uses Netflix Eureka.

The number of tactic-related microservices is described in Figure 5. Note that the numbers of microservices include functional as well as infrastructure microservices.

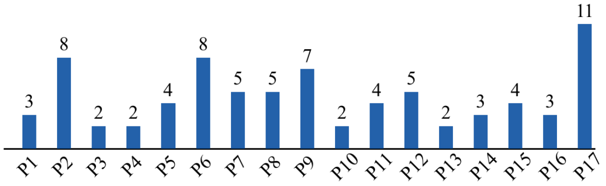


Figure 5: Tactics-related microservices per project

4.2 RQ2: Microservices availability tactics and documentation

Regarding the projects' documentation, Figure 4 describes which microservices availability tactics is possible to recognize. We realized that the most important source to identify microservices availability tactics are open and closed issues (see Table 4).

From a total of 4,188 issues, we collected 450 (≈ 10.87%) relevant issues (we considered issues posted by developers or architects, where keywords that define each tactic are manifested). Then, we omitted irrelevant issues and we obtained 122 issues where we find hints of tactics. Subsequently, we manually organized these

Table 4: Number of issues per project

Project	Open	Closed	Total	Relevant	Filtered
P1	17	46	63	21	8
P2	207	1053	1260	85	19
P3	6	1	7	3	0
P4	62	276	338	45	13
P5	*Not available*				
P6	139	1033	1172	82	24
P7	4	0	4	1	0
P8	40	575	615	87	23
P9	0	0	0	0	0
P10	17	17	34	5	5
P11	0	0	0	0	0
P12	5	1	6	0	0
P13	1	2	3	1	0
P14	1	5	6	2	0
P15	0	1	1	0	0
P16	1	12	13	7	5
P17	33	623	656	111	25
Total			4,188	450	122

issues by tactics and projects, and after merging similar issues, we obtained a final set of 21 issues from 7 OMbS (41% from the initial set of projects). Principally, most issues focused on describing the rationale of using T2 and T3 in specific scenarios.

4.3 Discussion

The results of our empirical study reveal that microservices-based systems are clearly concerned first and foremost with preventing faults, and secondarily with detecting them, instead of reacting or recovering from them. The previous observation coincides with the results obtained in our previous work regarding security mechanisms in microservices-based systems [22]. We detected that there is a lack of evidence on mechanisms that react or recover from attacks in the context of microservices.

In order to investigate why there is not enough analysis in the context of reacting and recovering from attacks as well as failures in microservices architectures, we examined academic [6] as well as industrial sources (InfoQ[12] and StackOverflow[13]). According with this review, there are three key security concerns with respect to microservices architectures:

Code reuse: The use of shared code and libraries can help migration to microservices, but it can also introduce lock-in problems and the need for propagation of patches to included components, possibly up to and including all the system's microservices.

Denial of service: Managing a set of services with multiple external entry points can be difficult. As the number of services grows, the probability of face a security incident also increase. This implies that microservice architectures are susceptible to denial of service attacks. Expensive API calls can produce multiple network hops around services, which may cause the system to attack itself.

Traffic between microservices: Microservices exchange information. If traffic happens in a segregated part of the network, it can be assumed that the risk of having a spy is reduced since it is usually behind a corporate firewall, hence less susceptible to man-in-the-middle attacks. However, when the microservices-based system is in an open cloud environment, this assumption is no longer valid, and besides adding microservices capabilities to handle encrypted traffic, which may affect the performance of the composing microservices.

These security concerns are mostly related to the prevention and detection of faults and attacks. However, to the best of our knowledge, we believe that there is a deficit of discussion about strategies that allow microservices-based systems to return to normal after suffering faults.

In the furtherance to address this gap, Toffetti et al. [26] discussed the possibility to manage recovery and/or react from failures strategies in microservices-based systems by introducing *self-healing* properties in the system. This feature is related to which steps should be executed to recover from a broken state. Generally, self-healing is implemented by an external system that watches the instances health and restarts them when they are in a broken state for a more extended period. Moreover, by placing components across different failure domains, microservices architectures can be resilient to failure and are able to guarantee that failed components will be restarted within seconds.

5 THREATS TO VALIDITY

In this section, we proceed to discuss the threats to the validity [29] of our study.

Threats to internal validity describe factors that could affect the results obtained from the study. We identified the following threats (i) bias on open source project selection, (ii) bias on microservices availability tactics identification, and (iii) bias on patterns and frameworks selection. In (i), we mitigated the threat by selecting GitHub, since it is a leading source code hosting platform. Furthermore, we used guidelines taken from other studies [1] and inclusion/exclusion criteria to obtain OMbS that were suitable for our research. Regarding (ii), we proposed 7 attributes to describe and characterize architectural tactics using documentation (see Section 2.3). Furthermore, we discussed our microservices availability tactics with practitioners with experience in microservices-based systems development in order to refine and improve the tactics. Finally, in (iii), brainstorming sessions, meetings with experts in academic conferences and interviews with practitioners allow us to mitigate this threat.

Threats to external validity are conditions that limit our ability to generalize the results of our experiment to industrial practice. To reduce threats to it, we described in Table 3 the URL of each OMbS used in this study. Moreover, in Table 2 we described the structure of architectural tactics definition that we used to identify them as well as the tools that helped in the source code analysis.

6 RELATED WORK

Regarding architectural tactics recovery, Mirakhorli et al. [18] [19] described a machine learning approach for discovering and visualizing architectural tactics in code. In the same line, Gopalakrishnan

[12] https://www.infoq.com
[13] https://stackoverflow.com

et al. [8] proposed a recommender system developed by building predictor models which capture relationships between topical concepts in source code and the use of specific architectural tactics in that code.

Concerning availability and microservices research, Wu et al. [30] proposed an extensible fault tolerance testing framework for microservice-based cloud applications based on the non-intrusive fault injection. From an architectural perspective, Parvizi-Mosaed et al. [21] proposed an automated evaluation method which composes architectural tactics and the patterns to measure the availability of software architectures.

Despite the significant contribution of previous studies, our study differs from the previous ones in the proposal of a format and structure to describe architectural tactics. Furthermore, previous studies focused on well-known architectural tactics, but they do not address how to identify and characterize emerging architectural tactics.

7 CONCLUSIONS

This article describes an empirical study related to the definition and identification of microservices availability tactics to support security design decision in microservices-based systems. We proposed a template to characterize architectural tactics (independently of the context), and we examined 17 OMbS aiming at distinguishing microservices availability tactics in source code as well as in the documentation. Key findings are (i) most microservices architectural tactics are focused on preventing faults, but we do not find tactics with the aim of recover or react from faults, and (ii) source code provide more evidence of microservices availability tactics than the documentation.

Future work points towards (i) to investigate the relationship and composition between microservices availability tactics and microservices patterns, and (ii) the creation of a recommender system that provides suggestions of architectural tactics and patterns in order to satisfy availability properties in microservices-based systems.

ACKNOWLEDGMENTS

This work was supported by Comisión Nacional de Investigación Científica (CONICYT) through grants PCHA/Doctorado Nacional/2016-21161005 and Centro Basal CCTVal.

REFERENCES

[1] M. Aderaldo, C. Mendonça, C. Pahl, and J. Pooyan. 2017. Benchmark requirements for microservices architecture research. *In Proceedings of the 1st International Workshop on Establishing the Community-Wide Infrastructure for Architecture-Based Software Engineering (ECASE'17)* (2017), 8–13. https://doi.org/10.1109/ECASE.2017..4

[2] F. Bachmann, L. Bass, and M. Klein. 2003. Moving from Quality Attribute Requirements to Architectural Decisions. *The Second International Workshop on From SofTware Requirements to Architectures (STRAW)* (2003), 122–129.

[3] F. Bachmann, Bass L., and Mark Klein. 2003. *Deriving architectural tactics: A step toward methodical architectural design.* Technical Report. Software Engineering Institute, Carnegie-Mellon University.

[4] Len Bass, Paul Clements, and Rick Kazman. 2013. *Software Architecture in Practice (3rd Edition).* SEI Series in Software Engineering.

[5] L. Bass, M. Klein, and F. Bachmann. 2000. *Quality attribute design primitives.* Technical Report. Software Engineering Institute, Carnegie-Mellon University.

[6] M. Dong, K. Ota, and A. Liu. 2015. Preserving Source-Location Privacy through Redundant Fog Loop for Wireless Sensor Networks. *2015 IEEE International Conference on Computer and Information Technology; Ubiquitous Computing and Communications; Dependable, Autonomic and Secure Computing; Pervasive Intelligence and Computing* (2015).

[7] Martin Fowler. 2002. *Patterns of Enterprise Application Architecture.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[8] R. Gopalakrishnan, P. Sharma, M. Mirakhorli, and M. Galster. 2017. Can latent topics in source code predict missing architectural tactics? *Proceedings of the 39th International Conference on Software Engineering* (2017), 15–26. doi:10.1109/ICSE.2017.10.

[9] Neil B. Harrison and Paris Avgeriou. 2010. How do architecture patterns and tactics interact? A model and annotation. *Journal of Systems and Software* 83, 10 (2010), 1735–1758. https://doi.org/10.1016/j.jss.2010.04.067

[10] S. Haselböck, R. Weinreich, and G. Buchgeher. 2017. Decision guidance models for microservices: service discovery and fault tolerance. *Proceedings of the Fifth European Conference on the Engineering of Computer-Based Systems* (2017), 4. doi:10.1145/3123779.3123804.

[11] Ralph E. Johnson. 1997. Frameworks = (Components + Patterns). *Commun. ACM* 40, 10 (1997), 39–42. https://doi.org/10.1145/262793.262799

[12] Rick Kazman. [n. d.]. Rapid Software Composition by Assessing Untrusted Components. ([n. d.]). https://insights.sei.cmu.edu/sei_blog/2018/11/rapid-software-composition-by-assessing-untrusted-components.html Accessed: 2018-11-26.

[13] J. Lewis and M. Fowler. [n. d.]. Microservices. A definition of this new architectural term. https://martinfowler.com/articles/microservices.html. [Online; accessed April 5th, 2019].

[14] G. Márquez and H. Astudillo. 2018. Actual Use of Architecture Patterns in Microservices-based Open Source Projects. *25th Asia-Pacific Software Engineering Conference (APSEC)* (2018), 31–40. doi:10.1109/APSEC.2018.00017.

[15] G. Márquez, F. Osses, and H. Astudillo. 2018. Review of Architectural Patterns and Tactics for Microservices in Academic and Industrial Literature. *Avances en Ingenieria de Software a Nivel Iberoamericano, CIbSE 2018* (2018), 71–84.

[16] G. Márquez, F. Osses, and H. Astudillo. 2018. Review of Architectural Patterns and Tactics for Microservices in Academic and Industrial Literature. *IEEE Latin America Transactions* 16, 9 (2018), 2321–2327.

[17] G. Márquez, M. Villegas, and H. Astudillo. 2018. A pattern language for scalable microservices-based systems. *12th European Conference on Software Architecture: Companion Proceedings (ECSA '18)*, Article 24 (2018), 7 pages. https://doi.org/10.1145/3241403.3241429

[18] Mehdi Mirakhorli and Jane Cleland-Huang. 2015. Detecting, Tracing, and Monitoring Architectural Tactics in Code. *IEEE Transactions on Software Engineering* 42, 3 (2015), 205–220. https://doi.org/10.1109/TSE.2015.2479217

[19] I. J. Mujhid, J. C. Santos, R. Gopalakrishnan, and M. Mirakhorli. 20‘7. A search engine for finding and reusing architecturally significant code. *Journal of Systems and Software* 130 (20‘7), 81–93. doi:https://doi.org/10.1016/j.jss.2016.11.034.

[20] S. Newman. 2015. Building microservices: designing fine-grained systems. *O'Reilly Media, Inc.* (2015).

[21] A. Parvizi-Mosaed, S. Moaven, J. Habibi, and A. Heydarnoori. 2014. Towards a Tactic-Based Evaluation of Self-Adaptive Software Architecture Availability. *International Conference on Software Engineering and Knowledge Engineering* (2014), 168–173.

[22] A. Pereira, G. Márquez, H. Astudillo, and E. B. Fernández. 2019. Security Mechanisms Used in Microservices-Based Systems: A Systematic Mapping (In press). *XLV Latin American Computer Conference (CLEI)* (2019).

[23] M. Richards. 2016. *Microservices AntiPatterns and Pitfalls.* O'Reilly Media.

[24] J. C. Santos, A. Peruma, M. Mirakhorli, M. Galstery, J. V. Vidal, and A. Sejfia. 2017. Understanding Software Vulnerabilities Related to Architectural Security Tactics: An Empirical Investigation of Chromium, PHP and Thunderbird. *International Conference on Software Architecture (ICSA)* (2017), 69–78. doi:10.1109/ICSA.2017.39.

[25] Davide Taibi. 2015. An Empirical Investigation on the Motivations for the Adoption of Open Source Software. *Tenth International Conference on Software Engineering Advances (ICSEA)* (2015).

[26] G. Toffetti, S. Brunner, M. Blöchlinger, and Edmonds A. Dudouet, F. 2015. An architecture for self-managing microservices. *Proceedings of the 1st International Workshop on Automated Incident Management in Cloud* (2015), 19–24. doi:10.1145/2747470.2747474.

[27] F. Ullah and M. A. Babar. 2019. Architectural Tactics for Big Data Cybersecurity Analytic Systems: A Review. *Journal of Systems and Software* 151 (2019), 81–118.

[28] M. E. Whitman and H. J. Mattord. 2011. *Principles of information security.* Cengage Learning.

[29] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén. 2012. Experimentation in software engineering. *Springer Science and Business Media* (2012).

[30] Na Wu, Decheng Zuo, and Zhan Zhang. 2018. An Extensible Fault Tolerance Testing Framework for Microservice-based Cloud Applications. In *Proceedings of the 4th International Conference on Communication and Information Processing (ICCIP '18).* ACM, New York, NY, USA, 38–42. https://doi.org/10.1145/3290420.3290476