How to design Future-Ready Microservices? Analyzing microservice patterns for Adaptability

João Francisco Lino Daniel
Free University of Bozen-Bolzano
Bozen-Bolzano, Trentino - Alto Adige
Italy
joao.daniel@student.unibz.it

Xiaofeng Wang
Free University of Bozen-Bolzano
Bozen-Bolzano, Trentino - Alto Adige
Italy
xiaofeng.wang@unibz.it

Eduardo Martins Guerra
Free University of Bozen-Bolzano
Bozen-Bolzano, Trentino - Alto Adige
Italy
eduardo.guerra@unibz.it

ABSTRACT

Microservices have become the de facto choice for large, complex systems, due to their drivers of cohesion and decoupling. According to this architectural style, the system is divided into small, independently deployable units, called microservices. In the literature, there are several architectural patterns for this style. Despite this, little is to be found focusing on extensibility. Further, in previous works, we analyzed the trade-offs involving some of those patterns, and among other findings, we identified some issues regarding extensibility in some cases. In the current work, we aimed to analyze how patterns affect the extensibility of microservice-based systems. We considered three specific approaches to extensibility: Microservice Internal Flexibility, Microservice Extensibility, and System Extensibility. We found 91 patterns in the literature, and analyzed 18 of them, of six different categories. The outcomes of this analysis are hoped to be useful for bringing extensibility as a key quality attribute when designing microservice-based systems.

KEYWORDS

microservices extensibility, microservices patterns, quality attributes analysis

ACM Reference Format:
João Francisco Lino Daniel, Xiaofeng Wang, and Eduardo Martins Guerra. 2023. How to design Future-Ready Microservices? Analyzing microservice patterns for Adaptability. In *28th European Conference on Pattern Languages of Programs (EuroPLoP 2023), July 05–09, 2023, Irsee, Germany.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3628034.3628046

1 INTRODUCTION

Microservices Architecture (MSA) is an architectural style in which the system gets divided into small, independently deployed parts, called microservices [8]. This style is driven by low coupling and high cohesion levels of the modules. For that reason, it has become the *defacto* architectural style for large, complex systems. Among other benefits, it enables each microservice to evolve independently of others, as well as changes to be added as new microservices instead of altering existing code.

EuroPLoP 2023, July 05–09, 2023, Irsee, Germany
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0040-8/23/07…$15.00
https://doi.org/10.1145/3628034.3628046

Nonetheless, MSA does not immediately yield these benefits. It is necessary to carefully deal with recurring issues, such as how to divide the responsibilities into microservices, or how to establish communication between different microservices, avoiding tightly coupling them.

However, practitioners are not alone in overcoming those issues. In the literature, there are studies about MSA patterns, conducting case analysis on usage of patterns [1], and also systematic mappings [11]. Also, there are works that classify patterns according to different groups [3, 7, 10] and act as a quick reference for professionals.

In previous works, we analysed the trade-offs of microservices patterns regarding size, coupling, and data sharing [5]. We also explored the relation between the patterns and architecture metrics [6]. From these experiences, we decided to have a broader study on extensibility, which comprehends the features of size, coupling, and cohesion [2].

In this work, our goals are expressed as the Research Question (RQ): ***"How does each pattern affect adaptability in a Microservices-based Architecture?"***. This RQ is broad and general, so it can be further specified. First, we define a small terminology that will help to understand the specification:

- **service** is a logical unit that contains a set of cohesive operations;
- **microservice** is a deployment unit, at runtime, it is a single process that might contain one or more services;
- **system** is a set of microservices that work together for a greater purpose; a system might be considered a part of a larger system (system with (sub)systems).

With these terms, we specify the RQ with the following Specific Research Questions (SRQ).

SRQ1) **Microservice Internal Flexibility**: *How does each pattern affect the ability of a single service to evolve internally without changing its interface?*
SRQ2) **System Extensibility**: *How does each pattern affect the ability of the system to incorporate new microservices?*
SRQ3) **Microservice Extensibility**: *How does each pattern affect the ability to add new features to an existing microservice?*

The rest of this document is organized as follows. In Sec. 2 is the methodology, where we detail the steps followed during this work. In Sec. 3 we present the patterns included in this work, with their description and sources. The analysis regarding the extensibility of each one, we present in Sec. 4. We discuss the results in Sec. 5, where we compile outcomes and lessons. We discuss the limitations and the their mitigations in Sec. 6. Finally, in Sec. 7 we conclude this work.

2 METHODS

2.1 Study Design

We systematically analyzed the patterns of two pattern catalogs: the Microsoft Cloud Design Patterns page[1], and Chris Richardson's Microservices Pattern Language[2]. We chose these two sources because, from previous works [4, 5], our perception is that the professionals broadly use these catalogs due to their accessibility, and both websites are maintained by experienced and reliable sources.

Initially, we gathered all the patterns present in the pattern languages. In total, we found 91 pattern documentation pages. This complete list counted with a variety of patterns, and even repeated patterns (sometimes under different names, but essentially the same solution). For this step, we did nothing regarding the aliases or purposes unrelated to the objective of this work.

The second step was to filter the patterns we would analyze in more detail. For that, we defined objective inclusion criteria, based solely on the text of the documentation of each pattern. The patterns selected were those that, somewhere in the text, contained at least one of the keywords related to our goals: ADAPTABILITY, ADAPTABLE, ADAPT, ADAPTATION, FLEXIBILITY, FLEXIBLE, EXTENSIBILITY, EXTENSIBLE, EXTEND, EXTENSION, COUPLING, COUPLE, COUPLED, LOOSE COUPLING, LOOSELY COUPLED, TIGHT COUPLING, TIGHTLY COUPLED, DECOUPLING, DECOUPLED, DEPENDENCY, DEPENDENT, DEPEND, MODIFIABILITY, MODIFIABLE, MODIFY, VARIABILITY, VARIABLE, PLUGIN, PLUG IN, PLUGGED IN, PLUGGING IN, PLUG-IN, OPEN SYSTEM, OPENED SYSTEM, OPENING SYSTEM. For this step, we wrote a JavaScript crawler application that would fetch the HTML code for the page of each pattern, and cross-reference it with the set of keywords, indicating which page contained each of the terms. Those patterns that had not hit with any of the keywords were ruled out of the process. From this point on, we worked with 43 patterns.

Aware of the fact that a simple word matching could bring potential false positives into the analysis, we ran the first round of applying the pre-analysis exclusion criteria. For each selected pattern, we manually found the occurrences of the keyword hit to understand their contexts and semantics. In those cases in which the term was found, but all the contexts were unrelated to the goal of the work, we also ruled them out. We were left with 24 patterns.

We then read all the patterns and highlighted fragments of the text that were evidence of a feature in the architecture of interest to our work. We analyzed each fragment and encoded it in regard to the feature.

Now, it was time to map the effects of the patterns to our SRQs, more specifically to each dimension an SRQ represents. For that, we created the Dimension Effect Criteria (DEC) for each relating the patterns with their effect on each dimension. We navigate in DEC's details in Subsec. 2.2.

To assess the DEC, and check if they were suitable for our purpose, we conducted a pilot examination. From the 24 patterns left, we got 8 at random and analyzed each according to those criteria. We discussed our results and felt they were adequate. Next, we followed through with all the 24 patterns. For each pattern, we highlighted relevant excerpts of the documentation text and mapped them with the features we were analyzing. This we called a "strike", to keep the count. A strike can have a positive or a negative impact on a feature. With that, we summarized how each pattern affected each dimension.

Finally, we went for a second round of exclusion, according to this post-analysis exclusion criteria: we would remove any pattern that was either an alias to another pattern or resulted in being neutral to all dimensions. After that, we ended with 18 patterns in the list.

2.2 Effects on the Dimensions

In this subsection, we detail the DEC. Table 1 presents the criteria for the effects on each dimension.

As an illustration of how these criteria were used, consider the "Shared Database" pattern. It says to use a single database shared among different microservices. Because of that, the adoption of this pattern *increases coupling on data structures.* As the criteria state, for instance, *coupling (data structure)* has an **inverted** relation to System Extensibility, in this example, the pattern would be considered to negatively impact this dimension. This individual fact would be registered as a single **negative** strike on System Extensibility. It is worth mentioning this analysis can lead to four types of effects on a dimension: "negative" is when all the strikes have a negative impact, "positive" is analogous but with a positive impact, "balanced" when there are mixed strikes, for both sides, and "neutral" is when there were no strikes in either direction

3 PATTERNS ANALYZED

In this section, we describe the 18 patterns that were analyzed. Also, we present a taxonomy where we group the patterns across 6 different categories: Communication, Data, Decomposition, Deployment, Interface, and Query. Fig. 2 illustrate this taxonomy.

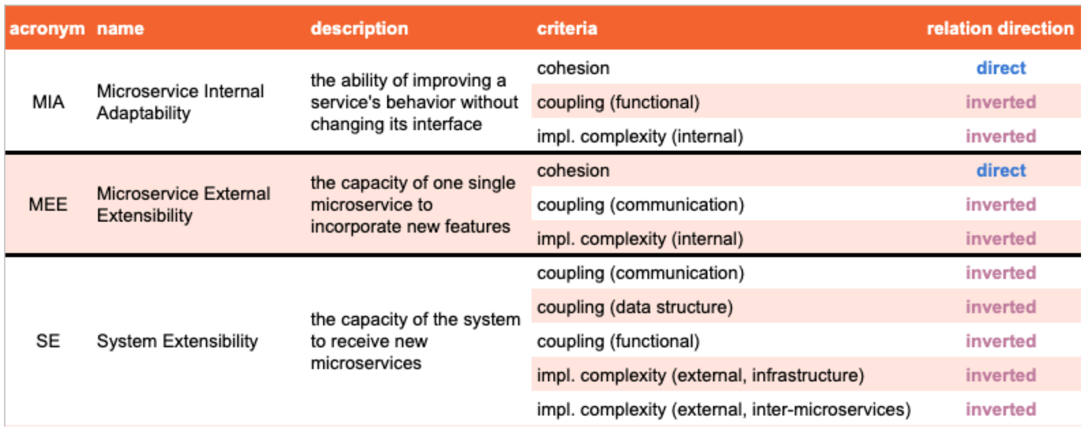
Ambassador Provides helper services that send network requests on behalf of a consumer service or application. https://learn.microsoft.com/en-us/azure/architecture/patterns/ambassador

Asynchronous Request-Reply This pattern allows a system to operate and process data in the background, freeing up user interfaces and systems from waiting for the completion of the task. It involves making a request that is processed asynchronously, often queued for processing and the reply is delivered once the processing is completed. https://learn.microsoft.com/en-us/azure/architecture/patterns/async-request-reply

Backends for Frontends Creates separate backend services to be consumed by specific frontend applications or interfaces https://learn.microsoft.com/en-us/azure/architecture/patterns/backends-for-frontends

Choreography In this pattern, each service involved in a business operation is designed to act independently and interact with other services through events. There's no central orchestrator dictating how each step of the business operation is carried out, instead, each service knows what step in the operation to execute

[1] https://learn.microsoft.com/en-us/azure/architecture/patterns/
[2] https://microservices.io/patterns/index.html



acronym	name	description	criteria	relation direction
MIA	Microservice Internal Adaptability	the ability of improving a service's behavior without changing its interface	cohesion	direct
			coupling (functional)	inverted
			impl. complexity (internal)	inverted
MEE	Microservice External Extensibility	the capacity of one single microservice to incorporate new features	cohesion	direct
			coupling (communication)	inverted
			impl. complexity (internal)	inverted
SE	System Extensibility	the capacity of the system to receive new microservices	coupling (communication)	inverted
			coupling (data structure)	inverted
			coupling (functional)	inverted
			impl. complexity (external, infrastructure)	inverted
			impl. complexity (external, inter-microservices)	inverted

Figure 1: The DEC - Dimension Effect Criteria

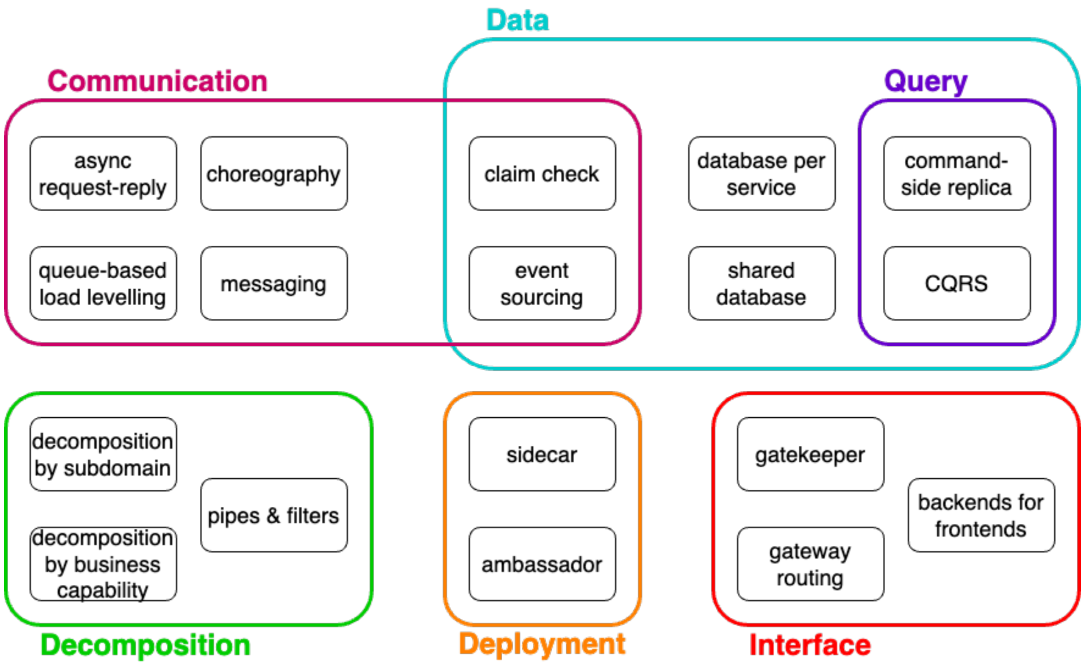


Figure 2: The patterns grouped into a taxonomy of categories

based on events it receives https://learn.microsoft.com/en-us/azure/architecture/patterns/choreography

Claim Check Splits a large message into a claim check and a payload to prevent the message bus and client from being overwhelmed https://learn.microsoft.com/en-us/azure/architecture/patterns/claim-check

Command-Side Replica This pattern involves separating the read and update operations for a data store. It maintains a replica of the data store on the command side (where updates are handled) which can be used to validate commands without needing to query the primary, read-side data store https://microservices.io/patterns/data/command-side-replica.html

Command-Query Responsibility Segregation Separates read and write into different models, using commands to update data, and queries to read data https://learn.microsoft.com/en-us/azure/architecture/patterns/cqrs

Database per Service Each microservice has its own database to ensure loose coupling and data consistency. https://microservices.io/patterns/data/database-per-service.html

Decompose by Business Capability Decomposes an application into services organized around business capabilities https://microservices.io/patterns/decomposition/decompose-by-business-capability.html

Decompose by Subdomain Decomposes an application into services based on subdomains within the business domain https://

microservices.io/patterns/decomposition/decompose-by-subdomain.html

Event Sourcing Preserves the history of aggregates by storing all state changes as a sequence of events. https://microservices.io/patterns/data/event-sourcing.html

Gatekeeper This pattern protects applications and shared resources from issues such as system overloads and malicious usage. It uses a gatekeeper component that validates and sanitizes requests before they reach a shared resource or service https://learn.microsoft.com/en-us/azure/architecture/patterns/gatekeeper

Gateway Routing Routes requests from clients to services using a gateway service https://learn.microsoft.com/en-us/azure/architecture/patterns/gateway-routing

Messaging Instead of services directly invoking each other, they communicate via messages. These messages are self-contained pieces of information that are sent over a message channel. A service sends a message into the channel, and that message is processed by another service that has subscribed to that channel. This decouples the services from each other, allowing them to evolve independently. https://microservices.io/patterns/communication-style/messaging.html

Pipes and Filters Breaks down a task that performs complex processing into a series of separate elements that can be reused https://learn.microsoft.com/en-us/azure/architecture/patterns/pipes-and-filters

Queue-based Load Levelling Uses a queue to level the load on a service and minimize the impact of peak load periods. https://learn.microsoft.com/en-us/azure/architecture/patterns/queue-based-load-leveling

Shared Database Multiple microservices use a shared database, each service can use data accessible to other services https://microservices.io/patterns/data/shared-database.html

Sidecar Deploys helper components of an application into a separate process or container to provide isolation and encapsulation. https://learn.microsoft.com/en-us/azure/architecture/patterns/sidecar

4 RESULTS OF THE ANALYSIS

To help summarize our results, and be useful for practitioners, we present two visual aids. Fig 3 has the purpose of giving the complete results for a particular pattern. It alphabetically places each pattern with its perceived impact, for each one of the dimensions analyzed.

4.1 How does each pattern affect the ability of a single service to evolve internally without changing its interface?

This dimension of Microservice Internal Flexibility represents how easy (or hard) it is to make internal changes to a microservice without impacting its interface. To make this kind of scenario more tangible, we exemplify. Suppose a change in the legislation requires additional recording of financial operations. In order to comply with this new regulation, the developers of the `CurrencyOperations` microservice will have to add a new layer of logging into their operations. Another: a team detected a bottleneck in the generation of a report, so they are considering improving the algorithm or changing the data model, to be more performant. The degree to which they can make this kind of change is what the Microservice Internal Flexibility dimension represents.

One criterion for analysing this dimension was how the patterns related to cohesion. There are benefits from the set comprising "Ambassador", "Backends for Frontends", "Command-Side Replica", "Decompose by Business Capability", "Decompose by Subdomain", "Gatekeeper", and "Pipes and Filters". The improves proposed by these patterns range from offloading peripheral concerns, to defining strong, business-driven boundaries between microservices, and specializing microservices by features. Also, there were no patterns negatively influencing cohesion.

Some patterns benefitted this dimension by its relation to functional coupling. On the positive side, there are "Backends for Frontends", "Choreography", "Command-Side Replica", "Gateway Routing", "Pipes and Filters", and "Sidecar". They did so, in broad terms, by reducing the direct dependency to other parts of the system. On the negative side, "Database per Service" and "Shared Database". In both cases, different microservices depend on each other to perform their data-related operations – in the first, it requires a direct interaction between the microservices; in the second, the data gets blocked to one when the other is accessing.

"Ambassador", "Command-Side Replica", and "Sidecar" reduce the internal implementation complexity. On all of these cases, this is achieved by separation of concerns into different parts of the code. Whereas, "Claim Check" and "Event Sourcing" add more processing to the microservice, either for querying data or for communicating with others.

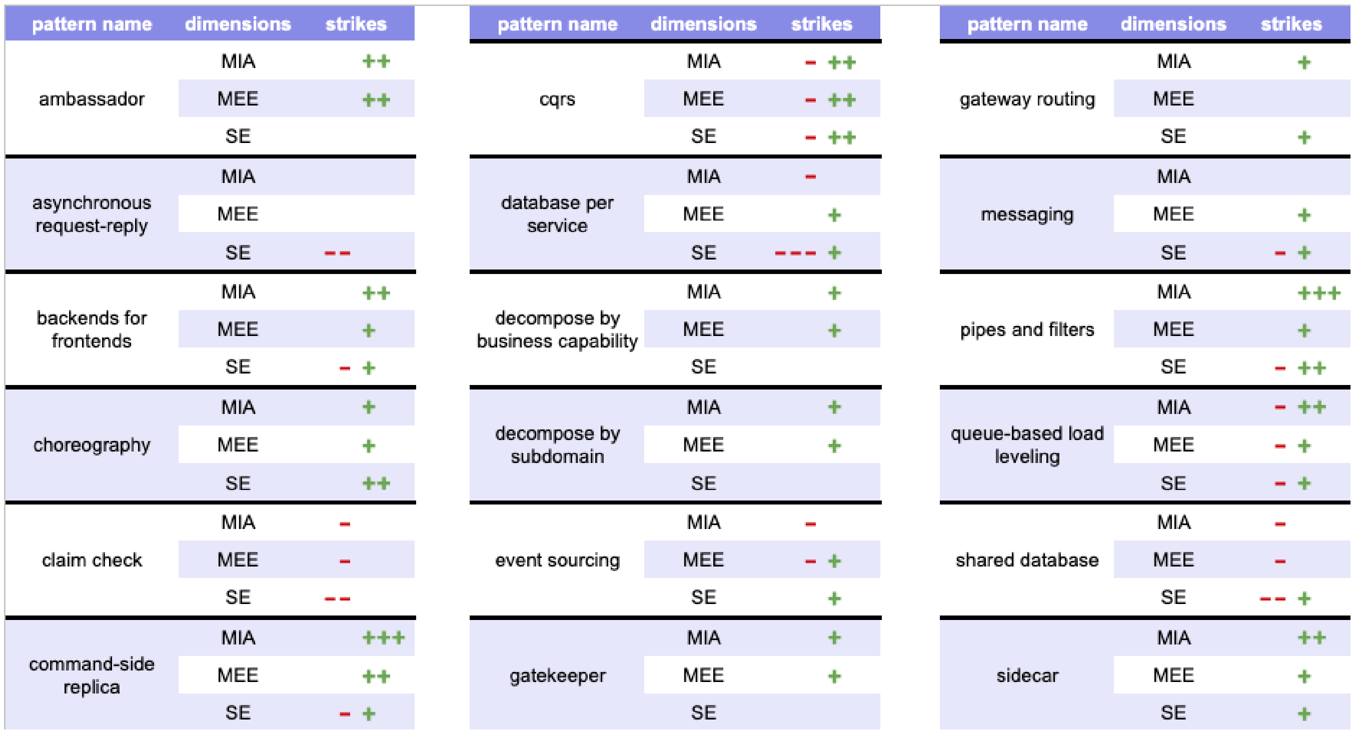
Finally, a few patterns have a balanced effect on Microservice Internal Flexibility. They are "Command-Query Responsibility Segregation" and "Queue-based Load Levelling", which both favor cohesion and reduce functional coupling, at the cost of additional internal implementation complexity.

We found "Asynchronous Request-Reply" and "Messaging" to be neutral in this dimension, i.e., we found no text fragments evidencing impacts in this dimension.

4.2 How does each pattern affect the ability of the system to incorporate new microservices?

System Extensibility focuses on the system (instead of individual microservices), and on how easy it is to add new microservices to it. We give two examples to illustrate these scenarios. When a company expands its domain, this requires new features to be developed and delivered. One way to deliver the feature is to deploy it into a new microservice. Another scenario is when the current architecture of the system is assessed, and it is decided to have a finer granularity of deployments, i.e., microservices need to get smaller without losing features. One approach to that is to split one microservice into two (or even more) new microservices. We represent the ability to do these kinds of changes with the System Extensibility dimension.

This dimension is affected by two groups of patterns: the first, positively by reducing coupling; and the second, negatively by increasing external implementation complexity. One the positive side, "Choreography", "Event Sourcing", "Gateway Routing", and "Sidecar" work on different ways of decoupling, from data structures, to



pattern name	dimensions	strikes
ambassador	MIA	++
	MEE	++
	SE	
asynchronous request-reply	MIA	
	MEE	
	SE	--
backends for frontends	MIA	++
	MEE	+
	SE	- +
choreography	MIA	+
	MEE	+
	SE	++
claim check	MIA	-
	MEE	-
	SE	--
command-side replica	MIA	+++
	MEE	++
	SE	- +
cqrs	MIA	- ++
	MEE	- ++
	SE	- ++
database per service	MIA	-
	MEE	+
	SE	--- +
decompose by business capability	MIA	+
	MEE	+
	SE	
decompose by subdomain	MIA	+
	MEE	+
	SE	
event sourcing	MIA	-
	MEE	- +
	SE	+
gatekeeper	MIA	+
	MEE	+
	SE	
gateway routing	MIA	+
	MEE	
	SE	+
messaging	MIA	
	MEE	+
	SE	- +
pipes and filters	MIA	+++
	MEE	+
	SE	- ++
queue-based load leveling	MIA	- ++
	MEE	- +
	SE	- +
shared database	MIA	-
	MEE	-
	SE	-- +
sidecar	MIA	++
	MEE	+
	SE	+

Figure 3: The effect of the patterns on the dimensions

functional, and communication. Whereas, "Asynchronous Request-Reply" and "Claim Check" propose solutions that add complexity to the implementation related to the infrastructure required for communication and data management.

On the balanced strikes, "Database per Service" offers challenges to external implementation complexity and to functional coupling, but favors decoupling of data structures. "Shared Database" compromises functional and data structure coupling, while reducing external implementation complexity. Then, "Backends for Frontends", "Command-Side Replica", "Command-Query Responsibility Segregation", "Messaging", "Pipes and Filters", and "Queue-based Load Levelling" favor decoupling – either communication, data structure, or functional –, at the expense of increasing external implementation complexity.

We found "Ambassador", "Decompose by Business Capability", "Decompose by Subdomain" and "Gatekeeper" to be neutral in this dimension, i.e., we found no text fragments evidencing impacts in this dimension.

4.3 How does each pattern affect the ability to add new features to an existing microservice?

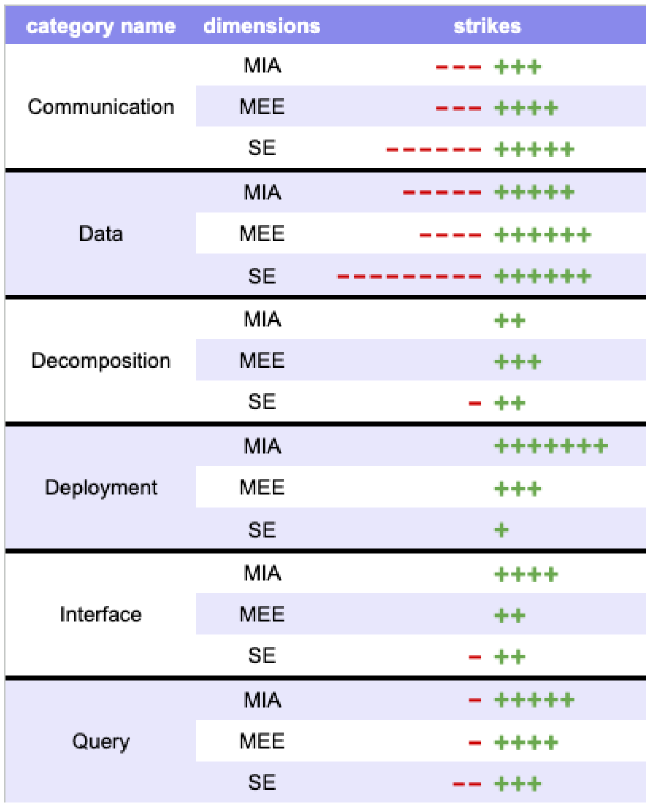
The third dimension sets the focus back into a single microservice. Microservice Extensibility represents the ability to add new features to an existing microservice. An example scenario that illustrates this: the product owners of a company designed a new feature that needs to be developed and shipped to the customers. This feature, in spite of being new, will have a high entropy with an existing one. So, one possibility to ship the new feature is to place it along with the existing one, in the same microservice. How much the existing microservice's code is prepared for this kind of injection of new code is what Microservice Extensibility represents.

In regards to cohesion, as a factor to affect the System Extensibility, there were patterns only in the benefit: "Ambassador", "Backends for Frontends", "Command-Side Replica", "Decompose by Business Capability", "Decompose by Subdomain", "Gatekeeper", and "Pipes and Filters". The reason for that is similar to before: strongly definition of boundaries, and offloading concerns.

Similarly, the only side of communication coupling affected was in the direction of reducing it. "Choreography" removes of a centralized orchestrator, while "Messaging" adds an indirection layer in the middle of the communication.

The last feature is the implementation complexity of the internal of a microservice. In here, "Ambassador", "Command-Side Replica", and "Sidecar" are beneficial, by favoring code reuse of peripheral tasks, or by removing the need of application code to handle the communication between parts. But, there is also "Claim Check" striking negatively, because the increase in the complexity for handling the payload of messages.

"Command-Query Responsibility Segregation", "Event Sourcing", and "Queue-based Load Levelling" have a balanced distribution of the strikes. They all have a negative impact because the increase of internal implementation complexity, but "Event Sourcing" favors communication decoupling, while the rest propose gains in cohesion.



category name	dimensions	strikes
Communication	MIA	--- +++
	MEE	--- ++++
	SE	------ +++++
Data	MIA	----- +++++
	MEE	---- ++++++
	SE	--------- ++++++
Decomposition	MIA	++
	MEE	+++
	SE	- ++
Deployment	MIA	+++++++
	MEE	+++
	SE	+
Interface	MIA	++++
	MEE	++
	SE	- ++
Query	MIA	- +++++
	MEE	- ++++
	SE	-- +++

Figure 4: The aggregated effect of the categories on the dimensions

We found "Asynchronous Request-Reply" and "Gateway Routing" to be neutral in this dimension, i.e., we found no text fragments evidencing impacts in this dimension.

5 OUTCOMES AND LESSONS

We analyzed the 18 patterns regarding, specifically, the dimensions we called Microservice Internal Flexibility, Microservice Extensibility, and System Extensibility. These patterns were classified into 6 different categories. Going a step further into the analysis of an individual pattern, we aggregated the strikes of the categories, to see how they affect the dimensions. Fig. 4 presents this information.

Although it is not possible to make any sense from comparing one category's results with another's, we can analyze them individually. For the purposes of this work, we can see that the patterns in the categories Decomposition, Deployment, and Interface are more generally beneficial to all the dimensions of Adaptability. This makes sense since they respectively favor the overall cohesion of the microservices, the maintenance of the microservices by offloading responsibilities, and the decoupling of the parts by abstracting the division.

The Communication, Data, and Query categories require attention when adopting their patterns. From our analysis, we see that, despite the high benefits, the patterns in the Query category also might compromise the Adaptability in some cases, in particular for the increases in the implementation complexity. A similar situation happens with Communication patterns and the implementation complexity, but in this case, the negative strikes were more numerous – which does not imply being worse in the impact; in reality, it only means there are more mentions of these types of effects.

The Data category is a separate one. Along with Communication, it is the biggest category in the number of patterns – both with 6 –, but it is the most complex one due to the intersections with other categories – namely, Communication and Query. Also, the strikes of Data were numerous for both sides on all dimensions. This all means we ought to treat the patterns in this category more carefully. The most frequent trade-off this category offers is favoring the decoupling of data structures, at the expense of adding complexity to the implementation. But Microservices is an architectural style that brings complex challenges to the process [8, 9], thus the solutions adopted can be inherently complex too.

To summarize the lessons learned in this work, we could highlight the following ideas:

- Decomposition, Deployment, and Interface patterns are important enablers of Adaptability in Microservices;
- the patterns of Query are solutions to be picked in a case-by-case manner – they favor Adaptability, but not for free;
- Communication and Data management are key aspects of the Adaptability in a Microservice system, so these patterns require a careful evaluation because they are valuable solutions but also impose some limitations.

6 LIMITATIONS

We understand this work is limited to our experiments, which might affect its broader applicability. In this section, we address these points and the strategies we followed to mitigate them.

The initial inclusion criteria of work matching against the documentation available can be considered weak. To mitigate this issue, we composed an extensive list of terms related to the goals of this work, that comprised "root" terms and a wide range of their derivations. For instance, "Adaptability" is a root term we included, but we also included its derivations "Adaptable", "Adapt", and "Adaptation".

Another possible issue would be regarding the quality of the documentation of the patterns. As our primary sources of pattern documentation were community-driven, it raises some questions, such as how well the patterns are described in the documents, or how representative of the patterns the texts are. For that reason, we selected two well-known collections, broadly used by the community, that are often updated. We used two different sources to avoid biases related to the aforementioned issues.

Finally, such a qualitative analysis is subject to personal biases. In this work, two of the authors independently analyzed the patterns according to the criteria, and then combined results. When there were divergences, we got together to discuss them, seeking convergence.

7 CONCLUSION

Microservices Architecture is a trending choice for complex systems, due to its emphasis on extensibility and other features it enables, such as independent evolution of its parts. From its patterns, in previous works, we identified some issues regarding increases in coupling level, as a means to achieve other qualities.

In this work, we addressed the RQ "How does each pattern affect adaptability in a Microservices-based Architecture?", an approach focusing on Extensibility as a quality attribute. We collected sources of patterns and analyzed 18 regarding Microservice Internal Flexibility, Microservice Extensibility, and System Extensibility. We identified most patterns positively influence Extensibility, as well as there are patterns that trade-off this attribute in favor of other aspects, or to solve more specific problems.

Furthermore, in this work, we present two main contributions. First, from the wide range of patterns, we compiled a set of 18 into a taxonomy that eases the understanding of the set. Second, we present the results of the analysis on the trade-offs involving Extensibility, highlighted by a diagram.

As future works, we understand there is value in adding patterns from other domains to help improve Individual Adaptability. Pattern languages such as from the Object-Oriented context can be used to drive internal implementations of a single service.

ACKNOWLEDGMENTS

We thank the contributions of Frank Frey, which were valuable for maturing and improving this work during the process of shepherding EUROPLoP-23. Also, we thank the members of our writers' workshop, in alphabetical order, Apostolos Zarras, Dionysis Athanasopoulos, Joseph Yoder, Marden Neubert, and Marwa Zeroual, whose comments and suggestions helped us understand how the paper was being perceived, and guided into this improved final version.

REFERENCES

[1] Armin Balalaie, Abbas Heydarnoori, Pooyan Jamshidi, Damian A. Tamburri, and Theo Lynn. 2018. Microservices migration patterns. *Software - Practice and Experience* 48, 11 (11 2018), 2019–2042. https://doi.org/10.1002/spe.2608

[2] L Bass, P C Clements, and R Kazman. 1997. *Software Architecture in Practice* (third edit ed.). Addison-Wesley.

[3] Kyle Brown and Bobby Woolf. 2016. *Implementation patterns for microservices architectures*. Vol. 22. 1–35 pages. https://dl.acm.org/doi/abs/10.5555/3158161.3158170

[4] Joao Francisco Lino Daniel, Alfredo Goldman, and Eduardo Guerra Martins. 2022. Are knowledge and usage of microservices patterns aligned? An exploratory study with professionals. In *2022 IEEE 46th Annual Computers, Software, and Applications Conference (COMPSAC)*. IEEE, 878–883. https://doi.org/10.1109/COMPSAC54236.2022.00139

[5] Thatiane de Oliveira Rosa, João Francisco Lino Daniel, Eduardo Martins Guerra, and Alfredo Goldman. 2020. A Method for Architectural Trade-off Analysis Based on Patterns: Evaluating Microservices Structural Attributes. In *Proceedings of the European Conference on Pattern Languages of Programs 2020*. ACM, New York, NY, USA, 1–8. https://doi.org/10.1145/3424771.3424809

[6] João Francisco Lino Daniel, Eduardo Guerra, Thatiane Rosa, and Alfredo Goldman. 2023. Towards the Detection of Microservice Patterns Based on Metrics. (2023). https://doi.org/10.1109/SEAA60479.2023.00029

[7] Microsoft. [n.d.]. Cloud Design Patterns. https://learn.microsoft.com/en-us/azure/architecture/patterns/

[8] Sam Newman. 2015. *Building Microservices - Design Fine-Grained Systems*. http://safaribooksonline.com

[9] Sam Newman. 2020. *Monolith to Microservices Evolutionary Patterns to Transform Your Monolith*. http://oreilly.com/catalog/errata.csp?isbn=9781492047841

[10] Chris Richardson. 2018. *Microservices Patterns* (1st editio ed.). Manning.

[11] Davide Taibi, Valentina Lenarduzzi, and Claus Pahl. 2018. Architectural patterns for microservices: A systematic mapping study. *CLOSER 2018 - Proceedings of the 8th International Conference on Cloud Computing and Services Science* 2018-Janua, Closer 2018 (2018), 221–232. https://doi.org/10.5220/0006798302210232