Information, Computing and Intelligent Systems, 2024, No. 5, 159 – 171

UDC 004.415.2 https://doi.org/10.20535.2786-8729.5.2024/316288

IMPROVING THE EFFECTIVENESS OF MONOLITH ARCHITECTURE TO MICROSERVICES MIGRATION USING EXISTING MIGRATION METHODS

Yaroslav Kornaha
National Technical University of Ukraine
"Igor Sikorsky Kyiv Polytechnic Institute", Kyiv, Ukraine
https://orcid.org/0000-0001-9768-2615

Hubariev Oleksandr *
National Technical University of Ukraine
"Igor Sikorsky Kyiv Polytechnic Institute", Kyiv, Ukraine
https://orcid.org/0009-0001-1028-4604

*Corresponding author: gubarev.alexandr@gmail.com

The theme of the transition from monolithic architecture to microservice is one of the key challenges of modern software engineering. This transformation allows for greater flexibility, scalability and adaptability of systems, but requires careful planning and consideration of numerous factors that affect the efficiency of migration. This study aims to improve the algorithm for determining the effectiveness of using methods for migrating monolithic systems to microservice architecture. Migration from monolithic architecture to microservice is a complex process involving significant technical and organizational challenges. Since monolithic systems often have a complex structure and relationships between components, the transition to a microservice architecture requires careful planning and selection of effective migration methods. The lack of a unified approach to assessing the effectiveness of different migration patterns makes the transition process difficult and risky.

The article is aimed at improving the algorithm for determining the efficiency of using migration methods from monolithic architecture to microservices. To do this, we compare existing migration patterns, such as the Strangler Fig Pattern, Branch by Abstraction, Parallel Run, Decorating Collaborator and Change Data Capture, according to the criteria: implementation time, test complexity, error risk, performance degradation and efficiency. The study uses methods of comparative analysis and quantitative evaluation of the effectiveness of migration patterns. For this, criteria are applied to assess the implementation time, testing complexity, possible risks, as well as the impact on system performance. In addition, scenarios are analyzed in which each template is most effective, which allows you to determine the optimal approaches to migration depending on the specifics of the project.

The obtained results allow not only a deeper understanding of the advantages and disadvantages of different approaches to migration, but also to form recommendations for choosing the optimal pattern, depending on the specifics of the system and business needs. The scientific novelty of the study is to create an algorithm that integrates these criteria to increase the efficiency of migration processes. The results of the work can be useful for software engineers, architects and managers planning the transition to microservice architecture, providing a structured methodology for evaluating and selecting migration methods.

Key words: microservice, monolith, distributed, architecture, transition.

1. Introduction

The migration of software systems from monolithic architecture to microservices is one of the most discussed and demanded topics in the modern field of software engineering. It directly addresses the problems of architectural software design and the management of complex information systems. Traditional monoliths, despite their structure, have significant shortcomings, which are manifested in the conditions of rapid scaling, constant changes in business requirements and the need for the rapid

introduction of new functions. In this context, the microservice architecture offers solutions that enable flexible, scalable and easily supported systems.

This article refers to the scientific field of software development and architectural modeling, as well as to related disciplines such as system integration and DevOps practices. The main problem of the study is to determine the optimal approaches and patterns for the gradual and safe migration of software systems from monolithic architecture to microservices. In particular, the question arises of choosing the right architectural pattern in the conditions of specific business requirements, infrastructure limitations and risks associated with the continuous operation of systems.

The relevance of the research is due to the need of modern organizations to create software that can quickly adapt to changes, easily scale and remain available.

So, research on the topic of migration from monolith to microservices using proven architectural patterns is relevant and practically significant. It is aimed at solving the key problem of optimizing the transition process, minimizing risks and increasing the efficiency of modern software systems. This will allow organizations to achieve strategic business goals, such as increasing the speed of innovation, reducing downtime and increasing competitiveness in a dynamic market.

2. Literature review and problem statement

The migration of monolithic architectures to microservices is a complex process that attracts significant attention from researchers and practitioners in the field of software development. This literature review highlights key works covering aspects of migration, in particular the patterns, strategies, challenges and tools used for a successful transition [1].

Work [2] describes microservices as small, independent services that interact through well-defined APIs. This work is considered a basic tool for understanding the concepts of microservice architecture.

Study [3] examines in detail the operational complexity arising from the distributed nature of microservices.

In the [4] article analyzes the evolution of microservices and their impact on the design of modern programs. The authors identify the main architectural patterns, such as Strangler Fig and Branch by Abstraction, as the most effective for gradual migration.The main thesis: microservices contribute to increasing modularity, but their implementation requires a significant restructuring of the architecture and development culture.

In the study [5] made overview of the most common patterns, including Parallel Run, Change Data Capture, and Anti-Corruption Layer. The article focuses on the need to take into account interdependencies in the monolith and describes approaches to reducing risks during the deployment of new microservices.

Work [6] focuses on technical challenges such as dependency management, the integration of old and new components, and the need for refactoring code.

Most works focus on the technical aspects of implementing individual patterns, but less attention is paid to assessing their impact on performance and scalability during and after migration.

Based on the analysis, it was found that the process of migration from monolithic architecture to microservice is accompanied by a number of challenges, among which the key ones are: ensuring data consistency, the complexity of the distribution of functional modules, infrastructure adaptation and organizational changes. Despite the significant advantages of the microservice architecture, such as scalability, flexibility and fault tolerance, the disadvantages associated with the increasing complexity of system development and operation remain significant.

The analysis showed that the problem of the lack of a universal approach to the development of a gradual migration strategy remains relevant, which would take into account the specifics of business processes, reduce technical risks and ensure the effectiveness of the transition period. Of particular interest is the development of methods that would simultaneously ensure: minimizing system malfunctions, increasing the independence of microservices, adapting the existing infrastructure without significant costs.

Migration is often accompanied by a significant waste of time and resources due to the lack of standardized approaches to automating the deployment, testing and monitoring of microservices.

And one of the biggest problems of monolith migration to microservices is the standardization of the process and the choice of the most effective approach for each individual project.

3. The aim and objectives of the study

The purpose of the study is to improve the algorithm for determining the effectiveness of using monolith migration methods for microservices.

Explore existing migration patterns such as Strangler Fig Pattern, Branch by Abstraction, Parallel Run, Decorating Collaborator, and Change Data Capture.

Compare patterns by criteria: implementation time, test complexity, risk of errors, performance degradation, efficiency.

Identify scenarios in which each pattern is most effective.

4. The study materials and methods of transition from monolithic architecture to microservices

4.1 Strangler fig pattern

The idea behind this approach is for our new system to be initially supported by and part of the existing system. That is, the old and new systems must coexist in the same environment. It gives the new system time to develop and adapt in order to be able to completely replace the old system or part of it in the future. The key advantage of this template is that it allows gradual migration to the new system. Moreover, it makes it possible to suspend and even completely stop migration at any step, taking advantage of the new system. Also, when we implement this idea for our software, we strive to ensure that every step can be easily undone, reducing the risks of new changes at every step.

If we consider this approach in the context of the transition from a monolith to a microservice architecture, the task will be to actually copy the code from the monolith, or re-implement the corresponding functionality. In addition, if the functionality in question requires state stability, then it is necessary to consider how this state can be transferred to a new service, and possibly back. To transfer functionality, you need to perform 3 steps (Fig.1.1) [7].

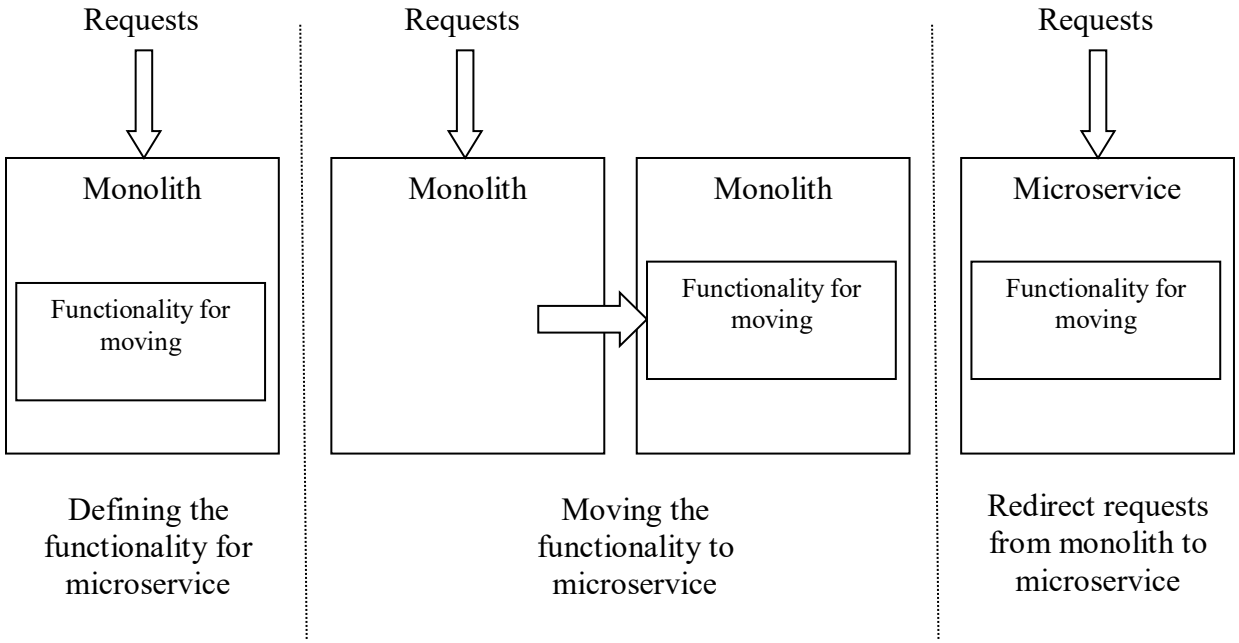

Fig.1.1 Algorithm of transition from monolith to microservices using Strangler fig pattern

At the first step you need to determine which parts of the system will be transferred and how it will help to solve the task, taking into account all the advantages and disadvantages of future changes. A list of results that correspond to what the business is trying to achieve should be created, and these results can be formulated in such a way as to describe the benefits to the end users of the system. And also, a list of criteria has been prepared that will help to understand whether the goal was achieved after the transfer of the selected functionality.

The second step is the implementation of this functionality in the new microservice. And after the new functionality is implemented in the new microservice, it is necessary to create an opportunity to redirect requests that are related to the implemented functionality from the monolith to the new microservice.

Until requests are redirected to a new microservice, the new functionality does not work, despite the fact that it is deployed in a production environment. This allows time for testing and gradual implementation of the new microservice into the existing system.

After the new service implements the same equivalent functionality as the monolith, you can consider using one of the deployment and testing patterns of microservices, such as Canary Deployment, Blue/Green Deployments, A/B testing. This will help test the operation of the new microservice as part of a system in a production environment with real users. If you consider release and deployment as two separate concepts, you can get the opportunity to test your software in the final production environment before using it, which reduces the risk of errors [2].

A key feature of this approach is not only that we can gradually bring new functionality to the new service, but that we can also roll back this change very easily if needed.

The advantages of this pattern include:

1. Allows you to smoothly transition from a service to one or more replacement services.
2. Keeps old services running when refactoring to updated versions.
3. Provides the ability to add new services and features when refactoring old services.
4. The pattern can be used to version the API.
5. The pattern can be used for interactions with legacy solutions that will not be updated.

The disadvantages:

1. Not suitable for small systems with low complexity and small size.
2. Cannot be used on systems where requests to the server system cannot be intercepted or routed.
3. A proxy or elevation level can become a single point of failure or performance bottleneck if it is not designed properly.
4. Requires a rollback plan for each updated service to quickly and safely return to the old way of working if something goes wrong.

4.2. Branch by Abstraction pattern

Considering the Strangler Fig Application pattern, it was about redirecting calls from the monolith to the newly created microservice. But, if the module that needs to be moved is deeper than the outer boundaries of the monolith and it is necessary to make significant changes to the system and to the modules that interact with this functionality. These changes can become a source of errors and incorrect operation of the system and rolling back such changes can be a much more difficult task.

Consequently, the main task is to be able to gradually make changes to the code base, as well as minimize failures for developers working on other parts of the code base. There is another pattern that allows you to gradually make changes to the monolith, without resorting to branching the source code.

The first step is to create an abstraction for the functionality you want to replace. Next, add changes to modules that use this functionality so that they can use the new functionality. The next step is to write a second abstraction implementation for the code, which will be allocated to a separate service. Next, you need to switch the abstraction so that a new implementation of the functionality is used. It remains only to remove the implementation that needs to be replaced and remove the abstraction (Fig. 1.2) [2].

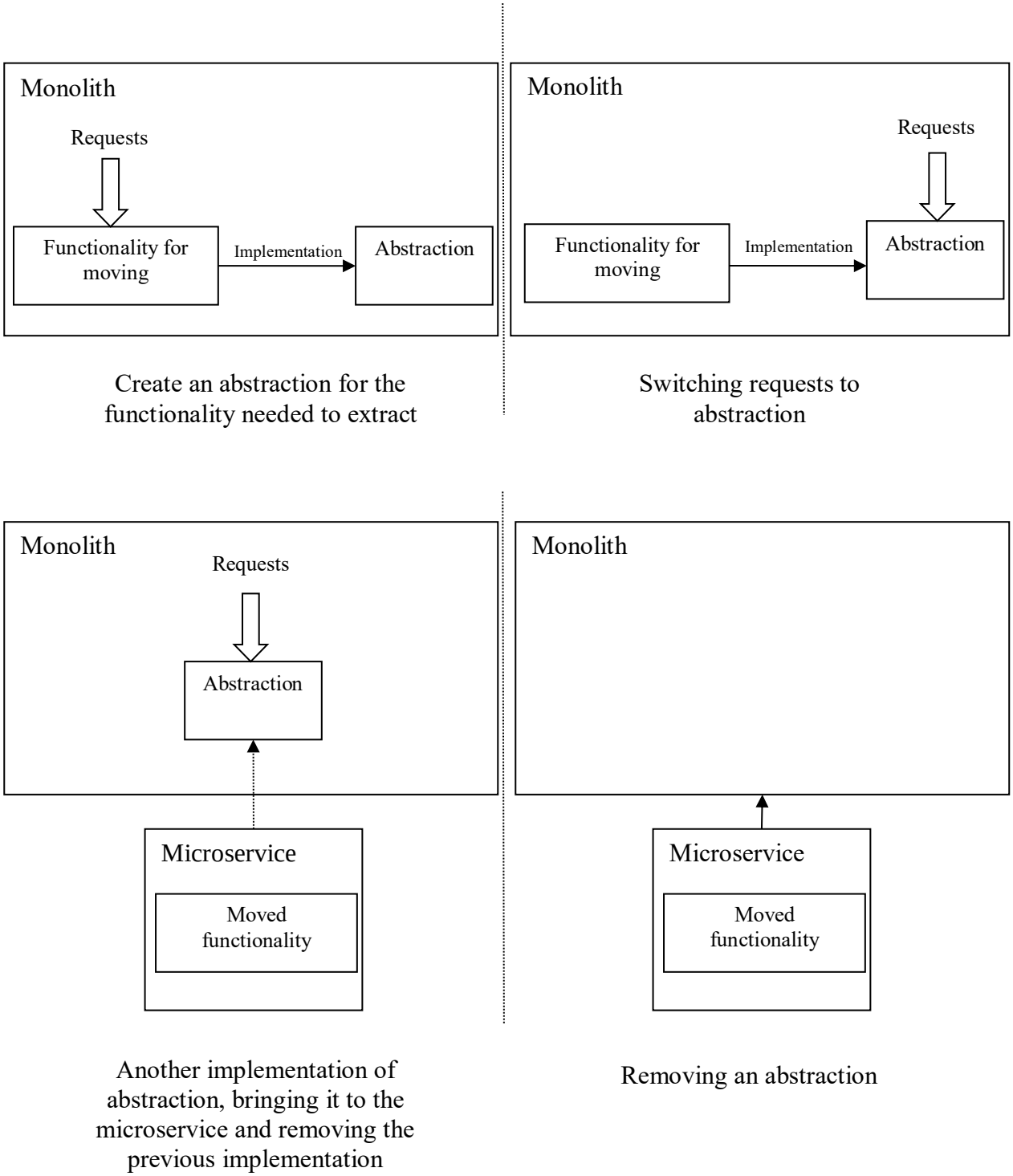

Fig.1.2. Algorithm for switching from monolith to microservice using Branch by Abstraction pattern

But how to roll back the changes if the newly created system does not work as expected? To do this, it is possible to leave both implementations of the functionality in the system and switch between them in case of failures and errors.

This adds complexity not only in terms of code, but also in terms of system research. In fact, both implementations can be active at any given time, which can make it difficult to understand the behavior of the system. If two implementations have a state, then the consistency of the data must also be taken into account. Since this pattern allows you to switch between implementations, this means that you will need a consistent common set of data that both implementations can access [2].

The advantages of this pattern include:

1. Allows incremental changes that can be undone if something goes wrong (backward compatibility).

2. Allows you to access functionality that is deep inside the monolith, in the case when it is impossible to intercept calls to it at the boundaries of the monolith.

3. Allows several implementations in the software system to coexist.

4. Provides an easy way to implement a backup mechanism using an intermediate validation step to invoke both new and old features.

5. Supports continuous delivery as the code runs all the time during the restructuring phase.

The disadvantages of this pattern include:

1. Not suitable when it comes to data consistency.

2. Requires changes to an existing system.

3. Can increase the cost of the development process, especially if the code base is poorly structured.

4.3. Parallel Run pattern

The previous two patterns that were described above allow old and new implementations of the same functionality to coexist in the production environment at the same time. Both of these methods allow you to perform either the old implementation in the monolith, or a new solution based on microservices. Also, to reduce the risk of switching to a new service implementation, these methods allow you to quickly return to the previous implementation. When using the Parallel Run pattern, instead of calling an old or new implementation, both are called, which makes it possible to compare the results to make sure that they are equivalent. Although both implementations exist and work, only one is considered the source of truth at any given time. As a rule, the old implementation is considered the source of truth, until the current check shows that we can trust our new implementation (Fig. 1.3).

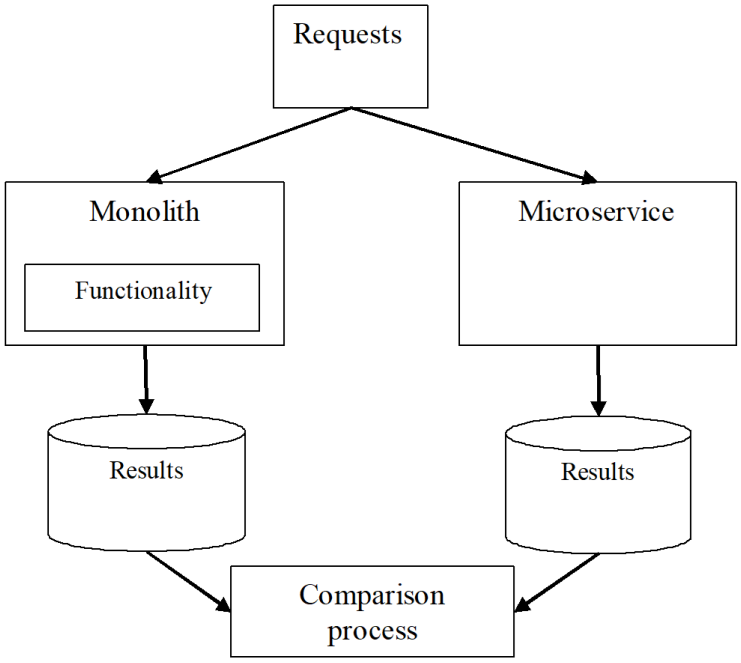

Fig.1.3. Algorithm for switching from monolith to microservice using Parallel Run pattern

This template has been used in various forms for a long time, but it is usually used to run two systems in parallel. This template can also be useful within the same system when comparing two implementations of the same functionality. This technique can be used to verify not only that a new implementation produces the same results as an existing implementation, but that it also works within acceptable non-functional parameters.

Advantages of the approach:

1. Testing with production data: makes it possible to test all possible use cases in a real environment.

2. Deployment takes place in stages: it is possible to activate functionality for individual cases.

3. Easy rollback: the new architecture is isolated from the old one. If, after activating the new service, the system does not work properly, the service can be easily deactivated and continue to use the old module.

4. Benchmarks: Enables you to compare old and new architectures to see which one works more efficiently.

Limitations:

1. Increased load: Given that requests received by the monolith are forwarded to the microservice, the load on all components increases, potentially doubling.

2. Nontrivial comparisons: Comparing results is not always an easy task. For example, comparing PDFs may be difficult due to different but minor metadata, or a change within http may result in different default response headers, or collections may have a different order.

3. It takes time to get the desired result: even if this approach leads to a smooth and safe migration, it takes a lot of time and effort to properly configure.

4.4. Decorating Collaborator pattern

In the event that it is necessary to trigger a certain behavior based on an event inside the monolith, but there is no way to change the monolith itself, you can use the Decorating Collaborator pattern. With this pattern, you can add functionality without changing the behavior of the monolith. The decorator here is used to make it seem that the monolith sends requests directly to services, although in fact the functionality of the monolith has not changed. Instead of intercepting these requests before they reach the monolith, the pattern allows the monolith to process this request. Then, based on the result of processing, the request will be sent to external microservices through any cooperation mechanism that will be selected (Fig. 1.4) [2].

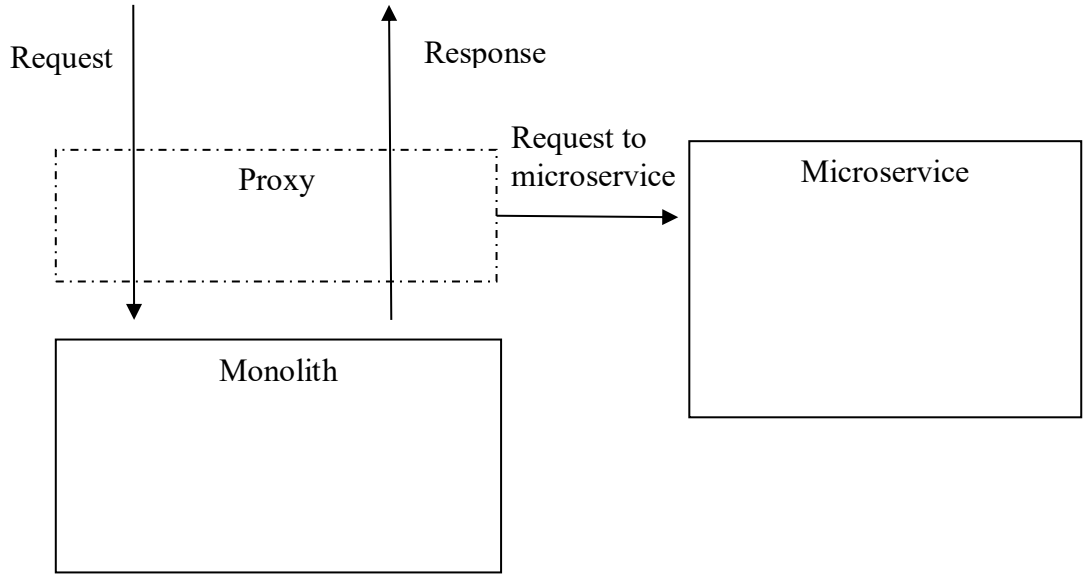

Fig.1.4 Algorithm for switching from monolith to microservice using Decorating Collaborator pattern

Advantages of the pattern:

1. New features are added without changes in the source code of the base object, which minimizes the risk of errors.

2. You can create dynamic combinations of behavior by combining several decorators.

3. The base object class remains closed to changes, but open to extensions through decorators.

4. Decorators are independent components that can be reused in different parts of the system.

5. Instead of creating new classes for each variation of behavior, combinations of existing decorators are used.

6. Each decorator can be tested separately, which simplifies the detection of errors.

Disadvantages of the pattern:

1. When using several decorators, it is difficult to understand exactly what changes were applied to the object.

2. May result in a complex call stack.

3. Each decorator is a separate object, which can increase memory consumption and complicate dependency management.

4. Risk of violation of the single liability principle (SRP):

In some cases, decorators may add too much heterogeneous functionality, making them difficult to maintain.

5. It is not always obvious what behavior the object will have at a particular time, which complicates the debug.

6. Conflicts can occur between decorators if they change the same behavior in different ways.

4.5. Change Data Capture pattern

With the change data capture pattern, instead of trying to intercept and respond to calls made to a monolith, we respond to changes made to the data warehouse. For the pattern to work, the basic interception system must be associated with the monolith data store (Fig. 1.5) [7].

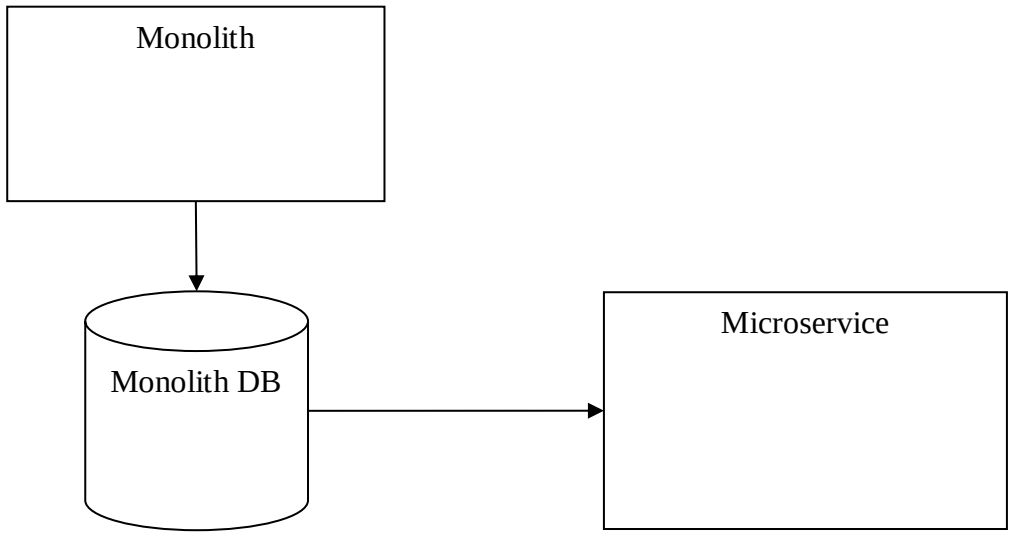

Fig.1.5 Algorithm for switching from monolith to microservice using Change Data Capture pattern

Different approaches can be used to implement data collection on changes, each of which has its own difficulties, advantages and disadvantages [8].

Database triggers. Most relational databases allow you to call a certain behavior when data changes (for example, 'INSERT', 'UPDATE' or 'DELETE'). How these triggers are defined, and what they can cause, depends on the database, but all modern relational databases support them anyway.

Triggers must be implemented in the database itself, just like any other stored procedure. There may also be limitations as to what these triggers can do. At first glance, this may seem quite simple. No need to work with any other software, no need to implement any new technologies. However, like stored procedures, database triggers can cause certain difficulties. The more triggers are created in

the database, the more difficult it is to understand how the system actually works. The problem is often maintaining and managing changes to database triggers [2].

Advantages:

1. Detects all types of changes: 'INSERT', 'UPDATE' and 'DELETE'.
2. Supported by most databases.

Disadvantages:

1. Affects output database performance through additional records.
2. Requires changes to the source database schema.

Transaction log pollers. Within most databases, there is a transaction log. This is usually the file in which all changes made to the database are written. To collect data on changes, the most complex tools, as a rule, use the transaction log. These systems operate as a separate process, and their only interaction with the existing database is through the transaction log. Here it is worth noting that only completed transactions will be displayed in the transaction log [2].

Important for using the transaction log is its format and it depends on different types of databases. Thus, which tools will be available will depend on which database is used. There are a number of solutions designed to display changes in the transaction log through messages that will be placed on the message broker; this can be very useful if the microservice is running asynchronously.

Advantages:

1. There is no additional load on the databases.
2. Detects all types of changes without having to modify the schema.

Disadvantages:

1. Lack of standardization of transaction log formats across vendors.
2. Target systems should recognize and ignore changes that have been rolled back in databases.

Batch delta copier. The most simplified approach is to write a program that regularly scans the database for data that has been changed and copies this data. These tasks are often performed using tools such as cron or similar tools. The main problem is to find out what data has actually changed since the batch collier was last launched. The design of the circuit may or may not make this obvious. Some databases allow you to view table metadata to see when data has been changed. But this is not a universal approach, and can only give time stamps of changes at the table level when it is necessary to have row-level information. You can add these temporary ones yourself, for example, add a special column to the tables that displays the time of the last change (for example, 'LAST _ MODIFIED' or 'LAST _ UPDATED'). Systems can then query this field to retrieve records updated since the last scan [2].

The advantages include:

1. Simple implementation.

Disadvantages:

1. Detects only soft deletions, not 'DELETE' operations.
2. Imposes a computational load on the source system through a full line scan.

5. Result of investigating on the effectiveness of methods of transition from monolithic architecture to microservices

To assess the effectiveness of migration patterns, a test environment was created based on a monolithic data management system using the Spring Boot framework. The microservice architecture was deployed on the Kubernetes platform using Docker containers.

The experiment was conducted in two stages: the first evaluated the speed of processing requests in the old monolithic architecture, the second – in a distributed system implemented using each of the patterns.

We will highlight the following key characteristics for analyzing the application of a particular approach:

1. Implementation time (T) – how quickly and with minimal effort you can apply the pattern.
2. Complexity of testing (S) – the amount and nature of work required to test the system after changes (conditional number of hours for testing).

3. Error Risks (R) – possibility of failures or malfunctions during implementation (% probability of errors). The error percentage (R) was calculated by the formula:

$$R = \frac{Count\ of\ errors}{Count\ of\ requests} \times 100\%. \quad (1)$$

4. System Performance (P) – Impact of changes on speed and stability (relative system performance reduction (%)). The decrease in productivity was calculated by the formula:

$$P = \frac{T_{new} - T_{old}}{T_{old}} \times 100\%, \quad (2)$$

where, Tnew – average time to perform operations in the new system (microservices), Told – average time of operations in the monolith.

Then the efficiency of using the method can be calculated by the formula (efficiency index):

$$E = \frac{1}{T \cdot S \cdot R \cdot \left(1 + \frac{P}{100}\right)}. \quad (3)$$

This formula reflects the relationship between key factors, which allows you to quantify the effectiveness of the method. Substituting the values of the corresponding parameters, the efficiency index provides a clear metric for comparing the effectiveness of this method with alternative approaches or basic scenarios.

This approach provides a systematic assessment, highlighting areas of improvement or confirming the suitability of the method to achieve the desired results.

Table 1 Comparative characteristics of methods of transition from monolithic architecture to microservices

Pattern	Implementation time (T)	Test complexity (S)	Risk of errors (R)	Performance degradation (P,%)	Efficiency (E)
Strangler Fig Pattern	4 months	5 days	8%	4%	0,6
Branch by Abstraction	2 months	4 days	10%	6%	1,17
Parallel Run	3 months	7 days	15%	15%	0,27
Decorating Collaborator	1 months	2 days	4%	8%	11,62
Change Data Capture	2 months	4 days	9%	6%	1,31

As can be seen from the Table 1 Decorating Collaborator has the highest efficiency due to its fast implementation, low testing complexity, and minimal risk of errors.

Branch by Abstraction and Change Data Capture have lower efficiency under the Decorating Collaborator due to the average level of testing complexity and implementation time.

Parallel Run has the lowest efficiency due to the high complexity of testing, significant performance degradation and risks of discrepancies in parallel systems.

Strangler Fig Pattern strikes a good balance between risk, performance and gradual migration, but takes longer.

6. Discussion of results on the effectiveness of using methodsof transition from monolithic architecture to microservices

Based on the defined criteria, conclusions can be drawn about the application of these patterns.

Strangler Fig Pattern is well suited for gradual migration with low risk, but takes longer.

Branch by Abstraction allows you to integrate new behavior quickly, but the complexity of testing increases due to working with abstractions.

Parallel Run is ideal for identifying differences between systems, but requires significant resources.

Decorating Collaborator is simple and flexible, but can reduce performance in complex systems.

Change Data Capture is useful for integrating new systems without stopping the current process, but requires careful testing of changes in data.

Each of the patterns considered can be useful in certain situations, depending on the specifics of the project, budget, time and risks, and the optimal choice depends on the specific needs and priorities in the modernization process.

The process of migration from monolithic architecture to microservices is a complex and multifaceted task that requires further research in a number of areas. Further research may focus on using statistical models and machine learning techniques to more accurately predict migration efficiency. Using data on past migrations, as well as analyzing their results using more complex algorithms, will improve the accuracy of forecasts for implementation costs and reduced productivity.

7. Conclusions

In this study, criteria were chosen and an algorithm for calculating the efficiency of patterns for the transition from monolithic architecture to microservices was proposed. Analysis showed that all five selected patterns have shown their effectiveness in certain contexts.

Each pattern provides a unique approach to solving the challenges of monolith-to-microservices migration. The most effective choice depends on the system's architecture, project requirements, and business constraints.

The Strangler Fig Pattern and Change Data Capture emerge as the most flexible and broadly applicable options, suitable for gradual and low-risk transitions. For high-availability systems requiring exhaustive testing, the Parallel Run pattern offers the best reliability. Meanwhile, Branch by Abstraction and Decorating Collaborator are advantageous for maintaining stability during complex migrations.

Based on the results obtained, you can identify scenarios in which each pattern is most effective:

Strangler Fig Pattern is effective in large organizations where the system has a complex architecture and migration requires gradual changes without stopping the entire system. This allows the transition from monolithic to microservice architecture over a long period.

Branch by Abstraction is most effective when you need to add new functions without disrupting the main functionality. This is suitable for medium and large projects where new technologies or components need to be introduced.

Parallel Run is ideal for critical projects where you need to check the performance and accuracy of a new system based on an old one. This applies in migrations that require high stability, for example in financial or medical systems.

Decorating Collaborator is best used when you need to add new features to an existing system without significant changes. This is suitable for projects where it is important to maintain the stability of the main functionality.

Change Data Capture is used for projects where it is important to synchronize data in real time, ensuring the continuity of business processes during migration.

The refined formula will be used to evaluate migration methods (e.g., Strangler Fig Pattern, Branch by Abstraction) against criteria such as implementation time, testing complexity, risk of errors, performance degradation, and overall effectiveness. This quantitative approach ensures a systematic comparison, leading to insights about the best practices and recommendations for specific migration scenarios.

The migration of a monolithic architecture to a microservice one is a strategic solution that can significantly improve the flexibility, scalability and sustainability of the system. However, this process requires careful planning, technical expertise and consideration of potential risks.

References

[1] S. Mooghala, “A Comprehensive Study of the Transition from Monolithic to Micro services–Based Software Architectures” Journal of Technology and Systems pp. 27–40, November 2023, https://doi.org/10.47941/jts.1538.

[2] S. Newman, Building Microservices: Designing Fine-Grained Systems, 2nd ed. O’Reilly Media, 2021.

[3] N. Ford and M. Richards, Fundamentals of Software Architecture: An Engineering Approach, O'Reilly Media, 2020.

[4] N. Dragoni, S. Dustdar, S. Giallorenzo, A. L. Montesi, M. Mazzara, F. Saez, and L. Troya, "Microservices: Yesterday, today, and tomorrow," in Present and Ulterior Software Engineering, 2017, pp. 195–216, https://doi.org/10.1007/978–3–319–67425–4_12.

[5] A. Balalaie, A. Heydarnoori, and P. Jamshidi, "Microservices migration patterns," in Softw. Pract. Exp pp. 1–24, July 2018, https://doi.org/10.1002/spe.2608.

[6] F. Tapia, M.A. Mora,W. Fuertes, H. Aules, E. Flores and T. Toulkeridis, “From Monolithic Systems to Microservices: A Comparative Study of Performance” in Appl. Sci. 2020, 10(17), 5797, https://doi.org/10.3390/app10175797.

[7] S. Eski and F. Buzluca, “An automatic extraction approach: transition to microservices architecture from monolithic application” XP '18: Proceedings of the 19th International Conference on Agile Software Development: Companion, May 2018, pp. 1–6, https://doi.org/10.1145/3234152.3234195.

[8] J. Fritzsch, J. Bogner, S. Wagner, A. Zimmermann, “Microservices Migration in Industry: Intentions, Strategies, and Challenges”, IEEE International Conference on Software Maintenance and Evolution (ICSME), June 2019, https://doi.org/10.48550/arXiv.1906.04702.

УДК 004.415.2

ПІДВИЩЕННЯ ЕФЕКТИВНОСТІ ПЕРЕХОДУ ВІД МОНОЛІТНОЇ АРХІТЕКТУРИ ДО МІКРОСЕРВІСНОЇ З ВИКОРИСТАННЯМ ІСНУЮЧИХ МЕТОДІВ МІГРАЦІЇ

Ярослав Корнага
Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського», Київ, Україна
https://orcid.org/0000-0001-9768-2615

Олександр Губарєв
Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського», Київ, Україна
https://orcid.org/0009-0001-1028-4604

Тема переходу від монолітної архітектури до мікросервісної є одним із ключових викликів сучасної інженерії програмного забезпечення. Ця трансформація дозволяє забезпечити більшу гнучкість, масштабованість і адаптивність систем, проте потребує ретельного планування та врахування численних факторів, які впливають на ефективність міграції. У даному дослідженні ставиться за мету вдосконалення алгоритму визначення ефективності використання методів міграції монолітних систем до мікросервісної архітектури. Міграція від монолітної архітектури до мікросервісної є складним процесом, що включає значні технічні та організаційні виклики. Оскільки монолітні системи часто мають складну структуру та взаємозв'язки між компонентами, перехід до мікросервісної архітектури вимагає ретельного планування та вибору ефективних методів міграції. Відсутність єдиного підходу до оцінки ефективності різних міграційних шаблонів робить процес переходу складним і ризикованим.

Метою статті є вдосконалення алгоритму визначення ефективності використання методів міграції від монолітної архітектури до мікросервісів. Для цього проводиться порівняння існуючих міграційних шаблонів, таких як *Strangler Fig Pattern, Branch by Abstraction, Parallel Run, Decorating Collaborator* та *Change Data Capture*, за критеріями: час реалізації, складність тестування, ризик помилок, деградація продуктивності та ефективність. У дослідженні використовуються методи порівняльного аналізу та кількісної оцінки ефективності міграційних шаблонів. Для цього застосовуються критерії, що дозволяють оцінити час реалізації, складність тестування, можливі ризики, а також вплив на продуктивність системи. Крім того, аналізуються сценарії, у яких кожен шаблон є найбільш ефективним, що дозволяє визначити оптимальні підходи до міграції залежно від специфіки проекту.

Отримані результати дозволяють не лише глибше зрозуміти переваги та недоліки різних підходів до міграції, але й сформувати рекомендації для вибору оптимального патерну залежно від специфіки системи та бізнес-потреб. Наукова новизна дослідження полягає у створенні алгоритму, що інтегрує ці критерії для підвищення ефективності міграційних процесів. Результати роботи можуть бути корисними для інженерів програмного забезпечення, архітекторів і менеджерів, які планують перехід до мікросервісної архітектури, надаючи структуровану методологію для оцінки та вибору методів міграції.

Ключові слова: мікросервіс, моноліт, розподілені, міграція, архітектура.