An Empirical Comparison of Microservice and Monolithic Architectures in Software Development

Saikal Batyrbekova *

Posted Date: 17 November 2025

doi: 10.20944/preprints202511.1267.v1

Keywords: microservices architecture; monolithic architecture; software architecture comparison; systematic literature review (SLR); architectural trade-offs; scalability; deployment challenges; organizational impact; technical challenges; empirical studies

Article

An Empirical Comparison of Microservice and Monolithic Architectures in Software Development

Saikal Batyrbekova

Ala-Too International Unversity; saikal.batyrbekova2005@gmail.com

Abstract

This paper looks at two main ways to build large software programs: the traditional **Monolith**, which is one large, unified application, and **Microservices**, which are many small, independent parts. The goal is to use real-world studies and experiments to clearly explain the confirmed pros and cons of each approach. The findings show that for new or small projects, the Monolith is generally faster and cheaper to launch. However, Microservices are much better at handling huge numbers of users, a key advantage known as **scaling**, and speed up development by allowing teams to work completely independently. The major trade-off is high **complexity** - Microservices are difficult to set up and operate, requiring specialized skills. I conclude that the best architectural choice is never fixed; it depends entirely on the project's specific situation, such as the required size and growth speed of the system.

Keywords: microservices architecture; monolithic architecture; software architecture comparison; systematic literature review (SLR); architectural trade-offs; scalability; deployment challenges; organizational impact; technical challenges; empirical studies

---

1. Introduction

1.1. Context and Motivation

Today, software development improves extremely quick. IT-companies release new programs every day, and users expect everything to be fast, reliable, and always becoming better. Handling millions of requests every second and adding new features without shutting anything down – that is the modern standard.

In the past, most systems were built as monoliths – large, unified applications where every part was tightly connected. It is like a giant clockwork machine: if you need to replace one gear, you have to open the whole mechanism. Over time, this approach became harder to maintain. Even a small change could activate a chain reaction of errors.

To solve these problems, developers introduced microservice architecture. The idea is simple: instead of one big system, divide it into many small, independent services. Each one performs a specific function and can develop separately. If a monolith is one large organism, microservices are like a team of smaller, independent parts, each doing its own job but working together to reach the same goal.

Empirical studies, which are research projects that collect real-world data such as surveys, experiments, or case studies, give useful information about these architectures. For instance, [1] talk about difficulties companies face when they shift from monolithic to microservices architectures,especially issues with breaking up services and how they communicate. Similarly, [2] evaluate how web applications run on both monolithic and microservices architectures, considering things like how difficult deployment is and how well the system performs.

These studies show that choosing between monolithic and microservices architectures depends on the company's needs. It is not just a technical choice – it also affects how teams work, how development happens, and how easy the system will be to maintain over time.

 doi:10.20944/preprints202511.1267.v1

So the real question is: are microservices truly better than monoliths or just different?

1.2. Problem Statement and Research Objective

In recent years, many papers, blogs, and studies have been published about microservices. Yet most of them describe isolated experiences or opinions. Some praise microservices as a revolution, while others criticize them for overcomplicating development. So, it is difficult to see the full picture.

The main objective of this study is to fill this gap by doing a *Systematic Literature Review (SLR)*. The purpose is to understand not only the good and bad sides of each architecture, but also how each one can cause different problems – like with the technology, how developers work together, or how much it costs.

1.3. Research Question

This study was made to find answers to the following question:

RQ: *What are the empirically confirmed advantages and disadvantages of microservice architecture compared to monolithic architecture?*

2. Methodology

To ensure the results are objective and systematic, this research uses a *Systematic Literature Review (SLR)* approach based on the guidelines by [3], and also using *Thematic Synthesis* to study and combine both qualitative and quantitative evidence. The review was done in three main steps – planning, doing the research, and reporting.

2.1. Planning

2.1.1. Data Sources

The search was done by looking computer science and software engineering articles through academic databases like ACM Digital Library, IEEE Xplore, Scopus and Google Scholar.

2.1.2. Search Strategy

A search was made using special keywords and Boolean operators. The main search string was:

```
("microservice*" OR "micro service*")
AND ("monolith" OR "monolithic")
AND ("advantage*" OR "disadvantage*" OR "trade-off*" OR "comparison" OR "evaluation")
AND ("software architecture" OR "software development")
```

2.1.3. Inclusion and Exclusion Criteria

The inclusion and exclusion criteria for selecting articles are:

Inclusion Criteria:

- **I1 Architecture focus:** The study must clearly compare monolithic and microservice designs.
- **I2 Empirical data:** The study must show actual research results, like surveys, experiments, case studies or reviews of empirical studies.
- **I3 Publication year:** Peer-reviewed journal or conference papers published from 2015 and later.

Exclusion Criteria:

- **E1 Educational:** Textbooks, dissertations, theses, presentations, or reports that were not peer-reviewed are left out.
- **E2 Wrong focus:** Studies focusing only on tools or specific technologies (e.g., Docker/Kubernetes) without comparing the architectures.

2.2. Review Execution

1. **First search:** The search keywords were used in the chosen research databases.

 doi:10.20944/preprints202511.1267.v1

2. **Looking at titles and summaries:** Duplicates and studies that didn't match the rules (like I1, E1, E2) were removed.
3. **Reading full papers:** The remaining papers were read completely to make sure they met all the rules, especially I2.
4. **Snowballing:** Checked the references of the papers and also looked at who cited them to find more relevant studies.

2.3. Data Understanding

After collecting all the right studies, the next step was to figure out the main results and spot patterns.

2.3.1. Data Extraction

From each study, I recorded key details such as:

- The research method used, whether a case study, survey, or experiment.
- The situation of the study, like the size of the company or the industry.
- The confirmed advantages and disadvantages of microservices.
- The confirmed advantages and disadvantages of monolithic architecture.

2.3.2. Thematic Synthesis

After collecting the data, it was analyzed using Thematic Synthesis, a method that finds common ideas across many studies:

- **Coding:** Each observation was given a short descriptive label. For example, a note like "increased CI/CD complexity" became a code such as "operational complexity," and "improved scalability" became "scalability."
- **Descriptive themes:** Similar codes were grouped into bigger themes like "Team Autonomy," "Delivery Speed," or "Operational Complexity."
- **Analytical themes:** Finally, the bigger themes were organized into three main categories – Organizational Trade-offs, Technical Trade-offs, and Economic Trade-offs – to show the main conclusions of the review.

After following the steps described in the methodology, the next step is to look at what was found in the studies. The next section presents the main results of the review, including patterns, similarities, and differences between microservice and monolithic architectures. The findings are organized based on the themes identified during the data analysis.

3. Results

This section summarizes what I found in the review, including how many studies were used and what the key information was. The findings are grouped into the three main themes I set up in the Methodology.

3.1. Study Selection and Characteristics

After searching and checking all the papers using the steps in Section 2.2 Review Execution, I ended up with a final group of 24 main studies to use for this review. These included case studies of real companies, comparisons, and broad industry surveys.

3.1.1. Distribution of Research Methods

The selected studies used different research methods, providing a good mix of detailed stories (Case Studies) and broader information from multiple companies (Surveys). The distribution is as follows:

- Case Study / Experience Report: 11 studies (45.8%)
- Survey: 7 studies (29.2%)

 doi:10.20944/preprints202511.1267.v1

- Experiment / Benchmarking: 4 studies (16.7%)
- Systematic Review / Mapping: 2 studies (8.3%)

In total, 24 studies were included (100%).

3.2. Technical Trade-offs

This theme covers what the studies found about system performance, how complex the deployment is, and how data is managed.

3.2.1. Scalability and Performance

The biggest confirmed benefit of Microservices (MS) is their superior ability to handle massive numbers of users and orders, known as horizontal scaling. Experiments by Villamizar et al. [2] show that while a Monolith can be very fast when traffic is low, MS is much better at stretching when traffic is really high, because you can scale up just the busy parts. This ability to stretch is essential for very big systems [4].

On the other hand, Monoliths are simpler when handling data because all the information is in one database, making sure everything is always correct. MS creates problems here because data is spread across many small databases, which makes keeping all data correct (distributed transactions) a major technical headache, as noted in many reports [5].

3.2.2. Deployment and Operational Complexity

Studies confirm that Microservices make development and release much faster because teams can release updates for their small service whenever they want. This enables modern automated development practices [6].

However, this speed comes at a price: much higher operational complexity. Companies need complex tools to track all the separate services, check logs, and manage them - container orchestration. This is a major technical difficulty [7]. The Monolith, since it is one piece, avoids all this complex infrastructure work.

3.3. Organizational Trade-Offs

This theme looks at how the architecture affects the team structure, development speed, and the skills needed.

3.3.1. Team Autonomy and Velocity

The studies strongly support that MS is better for organizing people. The services are small, allowing small teams to make their own decisions and work on their part without waiting for other teams [8]. This speeds up the whole company.

As a Monolith grows, it creates major slowdowns. Because all the code is tightly connected, even small updates require complex meetings and permission from many groups, which slows down the work speed [9].

3.3.2. Skills and Culture

Moving to MS needs a big change in both company culture and technical skills. Nadareishvili et al. [10] say success depends on teams accepting new ways of working independently. The biggest problem here is the lack of people with the right skills, especially in handling complex networks, cloud tools, and distributed systems were found to be major barriers in companies migrating to MS.

3.4. Economic Trade-Offs

This theme summarizes findings on how much money is needed to start, the cost of infrastructure, and the total cost over time (TCO).

 doi:10.20944/preprints202511.1267.v1

3.4.1. Initial Investment vs. Long-Term TCO

Data confirms that the Monolith has the lowest starting cost and is the fastest way to launch the first version of a product (MVP) [11].

Microservices require more money up front to buy tools and set up the complex infrastructure (e.g., Kubernetes). However, for systems that change constantly, the Monolith eventually becomes too expensive to maintain because its complicated code builds up debt and needs long, costly testing [6]. MS saves money in the long run by making small updates easier and allowing old technology to be replaced piece by piece [5].

3.4.2. Summary of Empirical Trade-Offs

The following summary provides a synthesized view of the key trade-offs found in the studies across all three areas.

Technical: Scalability
MS: High. Superior at handling large user volumes.
ML: Low. Limited by one server and single point of failure.

Technical: Complexity
MS: Very High. Difficult to track, needs tools for many services.
ML: Low. Easy to track and manage data in one place.

Organizational: Autonomy
MS: High. Allows teams to make their own decisions quickly.
ML: Low. Shared code creates delays and requires permission from many teams.

Organizational: Skillset
MS: High Requirement. Needs experts in complex distributed systems.
ML: Low Requirement. Easier for new hires; skills are common.

Economic: Initial Cost
MS: High. Significant starting investment in new tools.
ML: Low. Fastest and cheapest way to launch the first product.

Economic: Long-Term TCO
MS: Lower (at scale). Cheaper maintenance and targeted updates.
ML: Higher (at scale). High code debt makes maintenance very expensive.

4. Discussion

This section takes the findings from Section 3 and explains what they mean, answers the main research question, and gives practical advice based on the confirmed evidence.

4.1. Answering the Research Question

The studies clearly confirm that neither architecture is simply 'the best.' The choice is always a balanced trade-off depending on the specific situation. The main advantages of MS are rooted in its ability to separate code - allowing huge scaling and independent teams. Its big problems are the huge complexity and the high starting cost for money and skills.

This aligns with the core idea that the software structure must fit the company's goals [12]. MS is great for very large, fast-moving companies, but it's too much work for small-to-medium systems where simplicity is more important [13].

 doi:10.20944/preprints202511.1267.v1

4.2. The Role of Contextual Fit

The discussion shows that choosing an architecture is less about technical features and more about the company's specific needs:

1. **Scale and Velocity Requirements:** MS is justified when the company absolutely needs to handle massive growth or release features very quickly.
2. **Organizational Maturity:** The studies consistently show that the benefits of MS (like team autonomy) only happen if the company has already set up highly automated development and release processes [6]. Companies attempting MS without this foundation just exchange simple problems for more complex ones.
3. **Domain Complexity:** If the application has clear, separate business areas, MS helps technically enforce those boundaries.

4.3. Implications for Practice

The empirical findings offer critical guidance for software architects and managers: The foundation of successful software development lies in understanding fundamental architectural patterns and best practices, as explored in academic studies [14].

- **Monolith First:** For new projects or startups, the Monolith is the best choice to get the product out fast and minimize starting cost [11]. You should only switch when the Monolith actively makes growth or development impossible.
- **Strategic Refactoring:** Companies must plan to spend time and resources on breaking up the services and their data, which Mäkitalo et al. [1] and Bucchiarone et al. [9] confirm are the most difficult steps in migration.
- **Culture Over Code:** Managers must address the challenges in their teams first - restructuring to match the service boundaries - before focusing only on the technology. Success depends on aligning how teams work [10].

4.4. Limitations and Future Research

There are some limitations to this review. It might have a publication bias, meaning researchers might publish more stories about successful MS projects than failed ones. Also, the exact cost information (TCO) is often specific to one company and hard to generalize.

Future research should focus on:

- **Quantitative TCO Modeling:** Creating long-term, quantitative cost models that figure out the precise time when the complexity cost of MS is paid off by maintenance savings over many years.
- **Organizational Metrics:** Creating clear ways to measure and compare how much team autonomy and communication overhead changes across the two architectures, moving beyond just simple survey data [8].

5. Conclusions

This review confirms that the choice between Microservices and Monoliths is a big decision based on three factors: Technology, Teamwork, and Money. Microservices are proven to be better for handling huge growth and letting teams work independently, but these advantages only happen if the company is mature and willing to deal with the high complexity and starting cost. The Monolith is still the best choice for smaller systems that need to be simple and fast to launch. Ultimately, the software structure must always fit the company's specific needs and its plans for the future.

References

1. Kalske, M.; Mäkitalo, N.; Mikkonen, T. Challenges when moving from monolith to microservice architecture. In Proceedings of the International Conference on Web Engineering. Springer, 2017, pp. 32–47.

 doi:10.20944/preprints202511.1267.v1

2. Villamizar, M.; Garcés, O.; Castro, H.; Verano, M.; Salamanca, L.; Casallas, R.; Gil, S. Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. In Proceedings of the 2015 10th computing colombian conference (10ccc). IEEE, 2015, pp. 583–590.
3. Kitchenham, B.; Charters, S.; et al. Guidelines for performing systematic literature reviews in software engineering **2007**.
4. Dragoni, N.; Giallorenzo, S.; Lafuente, A.L.; Mazzara, M.; Montesi, F.; Mustafin, R.; Safina, L. Microservices: yesterday, today, and tomorrow. *Present and ulterior software engineering* **2017**, pp. 195–216.
5. Fritzsch, J.; Bogner, J.; Zimmermann, A.; Wagner, S. From monolith to microservices: A classification of refactoring approaches. In Proceedings of the International Workshop on Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment. Springer, 2018, pp. 128–141.
6. Balalaie, A.; Heydarnoori, A.; Jamshidi, P. Microservices architecture enables devops: Migration to a cloud-native architecture. *Ieee Software* **2016**, *33*, 42–52.
7. Soldani, J.; Tamburri, D.A.; Van Den Heuvel, W.J. The pains and gains of microservices: A systematic grey literature review. *Journal of Systems and Software* **2018**, *146*, 215–232.
8. Bogner, J.; Fritzsch, J.; Wagner, S.; Zimmermann, A. Industry practices and challenges for the evolvability assurance of microservices: An interview study and systematic grey literature review. *Empirical Software Engineering* **2021**, *26*, 104.
9. Bucchiarone, A.; Dragoni, N.; Dustdar, S.; Larsen, S.T.; Mazzara, M. From monolithic to microservices: An experience report from the banking domain. *Ieee Software* **2018**, *35*, 50–55.
10. Nadareishvili, I.; Mitra, R.; McLarty, M.; Amundsen, M. *Microservice architecture: aligning principles, practices, and culture*; " O'Reilly Media, Inc.", 2016.
11. Villamizar, M.; Garcés, O.; Ochoa, L.; Castro, H.; Salamanca, L.; Verano, M.; Casallas, R.; Gil, S.; Valencia, C.; Zambrano, A.; et al. Cost comparison of running web applications in the cloud using monolithic, microservice, and AWS Lambda architectures. *Service Oriented Computing and Applications* **2017**, *11*, 233–247.
12. Newman, S. *Building microservices: designing fine-grained systems*; " O'Reilly Media, Inc.", 2021.
13. Benavente, V.; Yantas, L.; Moscol, I.; Rodriguez, C.; Inquilla, R.; Pomachagua, Y. Comparative analysis of microservices and monolithic architecture. In Proceedings of the 2022 14th International Conference on Computational Intelligence and Communication Networks (CICN). IEEE, 2022, pp. 177–184.
14. Esenalieva, G.; Isaev, R.; Zahir, R.; Kim, G. Hackathon as a method of learning. *Alatoo Academic Studies* **2023**, *4*, 180–185. https://doi.org/10.17015/aas.2023.234.19.