Patterns in Design of Microservices Architecture: IT Practitioners' Perspective

Vadim Peczyński, Joanna Szłapczyńska
Faculty of Electronics, Telecommunication and Informatics
Gdansk University of Technology
Gdansk, Poland
vadim.peczynski@pg.edu.pl,
joanna.szlapczynska@pg.edu.pl

Anna Szopińska
Competency Center Digital
Sii Poland
Gdansk, Poland
aszopinska@sii.pl

Abstract—**The literature on Microservices Architecture (MSA) outlines a range of design blueprints as well as certain detrimental practices, reflecting the diverse architectural considerations inherent in MSA design. However, it remains unclear whether and to what extent the practitioners actually adopt the good practices. The study aimed to explore how MSA practitioners apply established patterns and how they address various architectural drivers. The advantages and disadvantages of these approaches were also examined. To achieve this, we conducted a survey on patterns in microservice design among a group of 77 MSA practitioners from IT companies worldwide. The survey shows a need for more accessible and standardised the MSA solutions supporting MSA design phase.**

Keywords-Microservices architecture, MSA, Survey, Patterns, MSA design

I. Introduction

The ideal starting point for a project is to build it with a monolithic architecture, utilising a single database and a single executable that can be easily run on a developer machine [1]. This type of architecture is structured with three primary layers: the client-side user interface, the server-side application, and a database. As the system grows, the maintenance of its architecture is becoming a challenge for developers and architects - all requests must be handled by a single process, and even a minor change triggers the deployment process for the entire application [2]. To overcome these disadvantages, a new type of architecture was introduced - Service-Oriented Architecture (SOA) [3]. SOA is an architecture designed with multiple services that collaborate with each other to provide the final set of functionalities. Each service is using a separate system process and promotes the re-usability of the software. This architecture also gives the possibility of replacing a service with another implementation as long as it keeps the same set of functionalities and communication interface. SOA usually still relies on a single database for the entire system, which ultimately results in the deployment of the entire application, and often uses the SOAP protocol for communication [3].

The Microservices Architecture (MSA) is an evolution of the SOA concept, offering greater independence through loosely coupled, small services that communicate via lightweight mechanisms such as: RESTful API or stream-based communication [4]. Microservices are designed for deployment in cloud environments, where their advantages simplify maintenance, enable autonomous scalability, and support independent deployment [5].

During the design phase of a MSA application, several challenges can arise, which require careful attention to ensure successful implementation. A primary challenge lies in determining the appropriate set of patterns to be employed during the implementation phase. Wrong architecture can lead to tightly coupled services, unnecessary fragmentation, or raising of technical debt [6], [7]. One possible approach involves supporting, balancing, and optimizing the Microservices architecture through the application of an appropriate set of design patterns. The goal of using design patterns in microservices design is to create a solution that satisfies the business's diverse needs while considering the various technical, operational, and financial factors at play.

The scientific literature provides a wealth of analyses and proposals for MSA design. However, it remains unclear to what extent these concepts are actually implemented by MSA practitioners in real world settings. This paper aims to address this gap by exploring the practices of a diverse group of MSA practitioners, primarily IT architects and software developers. We sought to understand the techniques they use for MSA design, the patterns and anti-patterns they apply. To achieve this, we designed and conducted a survey on MSA, involving 77 relevant participants in total from MSA professionals from IT companies around the world.

The rest of the paper is structured as follows. Section Background briefly outlines related literature and surveys on patterns and anti-patterns. Section Method introduces the MSA survey discussed in this publication, presenting the research questions, assumptions, detailing the groups of survey participants and examines potential threats to the validity of the survey. The following section presents results of the survey, categorised into areas: API Gateway, Circuit breaker, discovery mechanisms, transactional messaging, maintaining data consistency, querying and service observability. Section

Manuscript received July 14, 2025; revised September 2, 2025; accepted July 25, 2025. Published September 2, 2025.
Issue category: Special Issue on DSD/SEAA 2025 on Works in Progress (WiP) Session, Salerno, Italy, Sept. 2025
Paper category: Regular
DOI: doi.org/10.64552/wipiec.v11i1.101

Discussion interprets and discusses these findings and finally section Conclusions concludes the paper.

II. Related Works

Distributed systems, such as microservices, require a new set of technologies that must be integrated alongside the architecture. To manage initial setup costs, the use of new libraries and design patterns should be kept to a minimum [8]. In [9] the author analyses and describes diversified MSA design patterns applied to different levels of architecture such as communication, database, decomposition, discovery, deployment testing, and observability. In [3] the author extends the previous set of patterns with categories: reliability, scalability and security. Also, [3] proposes patterns focused more on human interaction and UI architecture (e.g. Micro Frontends, Central Aggregating Gateway, Backend for Frontend - BFF). Those patterns promote flexibility and loose coupling to enhance the development of large-scale systems.

Choosing the right set of patterns can be challenging and publications that address this topic can be found in [10], [11]. Also, the research community is increasing its attention around quality attributes (e.g. performance, scalability, security) in Microservices Architecture [4], [12]-[15] and the dependencies between microservices [16].

In [17], the authors also collect information about the usage of design patterns in MSA. They used the Likert scale to describe the use of patterns, and the comparison is discussed in the Discussion section. In [18], the authors propose queueing networks to obtain quantitative insights about seven performance-oriented patterns. Also in [11] the authors analyse the set of 14 design patterns on seven quality attributes during 9 semi-structured interviews. The set of patterns was chosen from the Azure Architecture Center [19].

The industry uses the patterns and strategies to improve the process of implementing the MSA, but many practitioners tend to overlook a critical aspect, the existence of anti-patterns and how they may evolve throughout the various phases of the transition. In [20] the authors describe eight anti-patterns and divide them into two categories: design and implementation. In [21] 19 anti-patterns are described and the research is also extended by adding visualization of these anti-patterns. In [22] the quality model based on 11 anti-patterns is proposed. It shows the need for solving this urgent issue in the form of a decision model lowering the impact of anti-patterns on overall MSA design.

III. Methodology

To guide the study, the research questions were formulated as follows.

- RQ1 - What are the most commonly used design patterns in MSA?
- RQ2 - What patterns are rarely used by practitioners?
- RQ3 - Is data consistency across multiple microservices maintained by design patterns?

To address these questions, we formulated a survey that was conducted among 77 participants from seven countries on three continents: Europe (Poland, Great Britain, Germany, Austria), North America (United States), and Asia (India, Afghanistan). The majority (82%) of the respondents work for companies with more than 1.000 employees. The participants work on the applications from sectors: IT (24%) followed by e-commerce (19%), finance (16%), engineering (11%) and others (30%).

In the survey, 92% of the respondents declared a programming role - out of which 29% are architects, 9% technical leaders and 54% software developers. The other 8% of the respondents are consultants, delivery manager, team leader, engineering manager, software quality (tester), director and chief procurement officer (CPO). The seniority of the participants is as follows: 83% of the respondents declared the level of senior knowledge, 14% declared the regular level of knowledge. Only 3% said they are at the beginning of their professional path (junior).

In the survey we focused on the design patterns commonly used side by side with MSA which can be found in the literature (Tab. 1). Patterns were divided into five groups related with their purpose:

- Communication and reliability,
- Discovery mechanism of Microservices,
- Transactional messaging,
- Maintaining data consistency,
- Observability and monitoring.

The first part of the questions focused on reliability (Circuit Breaker), external API patterns (API gateway) and querying techniques (CQRS, API Composition). These patterns are configured to establish reliable and secure communication with a distributed architecture.

The next part of the survey focused on the discovery mechanism of microservices. This mechanism is the most crucial topic for fault tolerance scenarios [23]. The services that are not working properly must be replaced by new instances and it is a typical action that improves the system's reliability. We asked our respondents if they are using the discovery mechanism in their applications, where the discovery of the services is placed (client-side or server-side), and if they use self- or third-party registration systems.

Transactional messaging was the subject of the next part. Each microservice maintains its own state and has its own database, if needed [3]. Several design patterns were introduced to overcome the problems with data consistency, distributed transactions, and eventual consistency. Transactional outbox (outbox pattern), message relay, and polling publisher are patterns that are responsible for establishing reliable communication between Microservices. Patterns were also added on the database layer where the transaction log miner uses the transaction log (transaction journal) and publishes each change as a message in message broker. We asked about usage of those patterns and which of them are used in the participants'

TABLE I. SYSTEMS. PATTERNS REFERENCE IN LITERATURE

Pattern	Referenced works		
	Richardson[9]	***Newman[3]***	***Newman[8]***
API Gateway	✓	✓	✓
Circuit breaker	✓	✓	
CQRS	✓	✓	
API Composition	✓		
Server-side discovery	✓	✓	
Client-side discovery	✓	✓	
3rd party registration	✓	✓	
Self registration	✓	✓	
Transactional outbox	✓		
Polling publisher	✓		
Transaction log tailing	✓		
Domain event	✓	✓	✓
Aggregate (DDD)	✓	✓	✓
Event sourcing	✓	✓	
Saga	✓	✓	✓
Log aggregation	✓	✓	✓
Application metrics	✓	✓	✓
Audit logging	✓		
Distributed tracing	✓	✓	✓
Exception tracing	✓	✓	
Health checks API	✓		
Log deployments and changes	✓	✓	
Correlation ID	✓	✓	✓

Data consistency in the distributed system is one of the most complex topics [24]. Maintaining distributed transactions when objects are constantly changing must be secure and consistent through all the databases involved in the process. We asked practitioners if they used any patterns to achieve this goal and which of these patterns are implemented in their applications.

The last part of the design patterns section of the survey was focused on observability and monitoring. In huge systems with MSA we need to rely on automation that will bring back all components in the case of any unexpected or faulty behaviour [23]. On the other hand, logs and monitoring services should provide us with documentation of such troublesome behaviour to improve the reliability of the system in the future. Detailed results of the survey are described in the next chapters.

We acknowledge the possible threats to validity related to the research method, the findings, and the strategies that were used to mitigate these threats. They are as follows:

- responses collected can limit their findings - 77 responses were received, the number might be increased if we redo the survey in future works;
- respondents may have different interpretations and understandings of MSA and the design patterns - graphics describing patterns were provided and open option was added in most of the questions to give space also for other answers;
- lack of clarity of the questions - four pilot surveys were conducted with system architects with extensive experience in MSA, language of some of the questions was improved;
- responses from those who were not involved in designing the Microservices systems - by using branching, we closed some of the questions to those respondents without experience in MSA;
- some of the design patterns might have not been mentioned in the survey - open answer was added for any other pattern that was not mentioned.

IV. RESULTS OF THE SURVEY ON MSA PATTERNS AND ANTI-PATTERNS

The study focuses on analysing design patterns commonly used in MSA. The main problems which can be encountered during work with Microservices are: distributed transaction, discovery and reconnection mechanisms, data consistency, and querying.

Communication and reliability. In MSA large monolithic applications are divided into smaller modules (microservices). In this approach the potential points for a cyberattack is bigger, because each Microservice has its own interface for the communication. To mitigate some of the potential risks, the API Gateway pattern was introduced [15]. In addition, it solves problems with cross-platform compatibility and inconsistent issues with microservices call standards [25]. This pattern can also be extended into a Backend for Frontends approach (using multiple API Gateways), which further enhances its versatility. The API Gateway pattern is commonly used in the projects of the respondents (73%). More than a quarter (26%) is not using it in their projects. One participant decided to not answer this question. This pattern's popularity is also evident when examining its usage broken down by project role (Fig. 1).

Circuit breaker is used to improve the resiliency of the MSA. During communication between Microservices Circuit Breaker detects faults and protects the system from cascading failures [26]. It works like a fuse, and when failures consecutively cross the threshold, a circuit breaker will stop the downstream request (open state) for a certain period. After that period, the circuit breaker allows part of the test calls (half-open state) and resumes normal operation (close state) until these calls succeed [4]. The Circuit Breaker is not as commonly used in participant projects in comparison to the API Gateway even though its complexity is compensated by already existing implementations within libraries (e.g. Polly, Resilence4j). It is used in 36% of the projects, 61% declares that they are not using it, and 3% (two participants) did not answer the question.

In MSA the information is scattered between different databases belonging sometimes to hundreds of microservices. In the monolithic architecture, a single database can provide the dedicated views serving data that the user is looking for. One of the challenges in MSA is to handle querying the data across the whole system. One of the potential solutions to this problem is by using API Composition pattern. The pattern provides a simple method to query the data in MSA [17]. The API Composer is a central point of the querying system which knows which microservice endpoint should be called to get the data. Potentially a front-end client could be an API Composer, but due to firewall restrictions and network limitations, it is better to use API Gateway as an API Composer (API Gateway is an internal part of the server solution). This pattern is quite simple and intuitive for querying in MSA. However, it also has its drawbacks such as higher costs of the infrastructure (calling of multiple services each time when data is needed), risk of lower availability (API Composer and all involved microservices need to be available for a query), and potential inconsistencies in transactional data. A more detailed description of this pattern can be found in [9]. The second approach can be CQRS - a pattern that separates read from write operations by querying different databases and keeping them in sync using a dedicated strategy (e.g. Event Sourcing or Relational Database Management System trigger with a special flag to mark data as 'dirty') [10]. This pattern can also be implemented as a single centralised service with dedicated views updated by changes in other databases. The advantages of using CQRS are as follows:

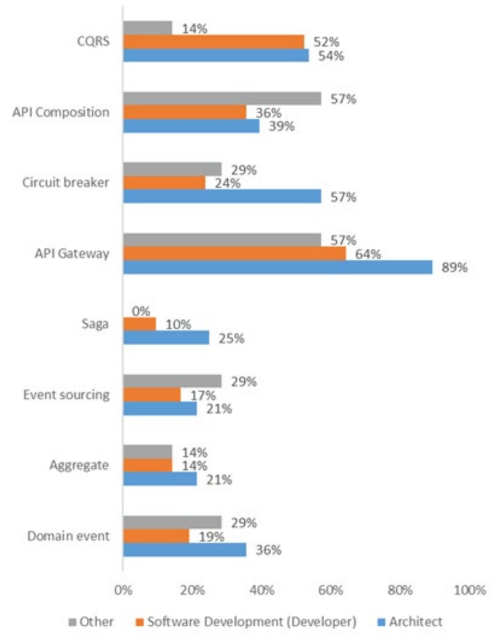


Figure 1. Resilience, communication and data maintenance patterns divided by role in the team among all particpants

- efficient implementation of querying in MSA (one single DB with dedicated views),
- efficient implementation of diverse queries (different databases types can be easily handled),
- can be connected with Event Sourcing,
- improves separation of concerns.

Using CQRS can also have disadvantages related to that:

- system architecture is more complex,
- replication lag needs to be taken into consideration.

A detailed description of this pattern and its advantages and disadvantages can also be found in [9].

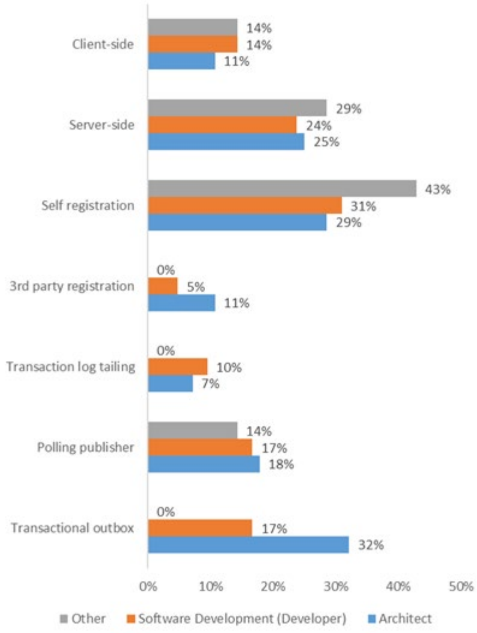
According to survey respondents, CQRS is the most popular approach for querying in the MSA (41%). API Composition is used in 32% participants' projects. There are also respondents who do not use any pattern (23%) and left the answer to this question blank (3%). There is also one other response: "The system uses the REST in communication with the user", which may indicate the usage of API Composition.

Discovery mechanism. The Microservices' environment is very dynamic - virtual machine instances are started and stopped due to failures and scalability features of the MSA. The discovery mechanism uses the Service Registry to store all available instances in the system and helps routing application traffic [23]. The next question was obligatory for all participants and the following two were answered only if the answer was 'Yes' to the first one. The service discovery mechanism is used by 38% of the participants, which is quite low number if we consider the dynamic nature of the MSA - new instances are added to the system when others are shut down within sometimes seconds.

In service discovery, we can use two major approaches: client-side and server-side [9]. Client-side is using Service Registry to get all running instances and using load balancing algorithm (e.g. round-robin or random) is choosing the server which will be used. The main advantage of this approach is the possibility of using multiple platforms (e.g. Kubernetes and an in-house solution with local data centre servers). The disadvantages of this solution are: handling of service discovery mechanisms on client side (especially hard with different technological stack in each microservice), configuration, and maintenance of the service registry as part of MSA. The second approach is to use server-side (platform) discovery. In this approach, the client calls a router, which is load balancing the traffic to all registered services (after querying the service registry). The main advantages are: client code is simpler due to the fact that it does not need to deal with discovery and use one of the available solutions e.g. Azure Load Balancer, Amazon Elastic Load Balancer. The disadvantages are: maintenance of the router (if it is not cloud based), router needs to support the communication protocols (e.g. HTTP/S, gRPC) also more network hops are required in comparison to client-side [27]. The server side is the most popular approach among service discovery users (66%) and the client-side implementation is declared by 34%.

The second part of service discovery is the registration mechanism. It can be implemented in two forms: self-registration and third-party registration [9]. In the self-registration each instance should inform the service registry that it is up and running. The advantage is that each service knows its own state and can give more information than up or down, e.g., starting, available, warm-up [28]. As a disadvantage of this approach we can point: coupling to service registry, each instance needs to implement service registration logic, faulty instance (running, but not able to handle requests) has problem with unregistering from service registry. The second approach – a 3rd party registration - is adding 3rd party registry which is responsible for registering and unregistering a service. Advantages of this approach are the following: the service code is less complex than in self-registration. The registry can also perform periodic health checks. The disadvantages are simplified state knowledge (running or not running) and having another component in the architecture (which sometimes must be additionally installed) [29]. The majority of service discovery users (83%) prefer to use self-registration and other users declare using 3rd party registration (17%).

Service discovery does not appear to be widely adopted also when we analyse it by the different participant roles. While the mechanism is inherently complex to implement [9], its adoption can be significantly simplified by leveraging existing solutions such as Kubernetes, AWS Service Discovery, and Consul. Among those who do use it, self-registration and server-side approaches are the most common, with usage distributed fairly evenly across all three groups (Fig. 2).



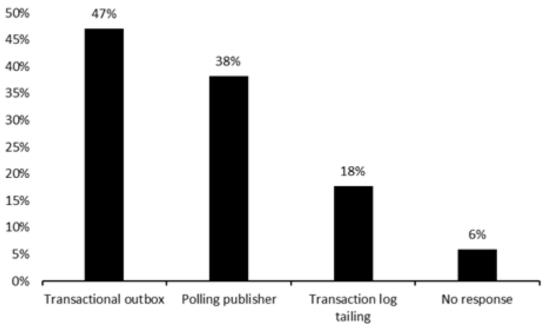


Figure 2. Service discovery and transactional messaging patterns usage divided by role in the team among all participants

Figure 3. Transactional patterns used in participants' projects

Transactional messaging. In MSA each microservice should maintain its own state, and microservices should avoid sharing the database and instead have a database per microservice [3]. This leads to possible problems with data consistency, ACID transactions, and supporting multiple denormalization [17]. Several design patterns were introduced to overcome these issues. One of them is a transactional outbox (outbox pattern) used in databases to store all messages in a table called OUTBOX. The atomicity of the operation is kept due to the fact that the transaction is local. Another pattern is message relay, where messages are read from the table and published to the message broker. The next design pattern is focused on the message moving from the database to the message broker. The polling publisher periodically searches the database for waiting messages and publishes them on the message broker. Finally, the messages are removed from the database. A more sophisticated approach assumes using the transaction log (transaction journal). Each database operation there is stored as an entry in the transaction log. The transaction log miner reads the transaction log and publishes each change as a message in the message broker. This approach can be implemented for relational databases or NoSQL databases. Detailed descriptions of these patterns can be found in [9].

Respondents were asked if they use any kind of transactional messaging pattern. Almost half of the participants (44%) declare that they use these patterns in their projects.

The next question was only available for those participants who answered 'Yes' in the previous question. The respondents were asked which patterns are actually used in their projects, with the possibility of selecting multiple patterns. The most popular pattern is the transactional outbox (47% of 34 responses), but was sometimes not marked together with the polling publisher or the transaction log tailing (47% of the answers), which transactional outbox relies on. The respondents prefer to use the polling publisher (38%) than the transaction log tailing (18%). Two respondents decided not to mark any answer even though there was some other option (6% out of 34 answers). The results are visualised in Fig. 3.

The transactional outbox is used most frequently by architects, which may indicate that it is primarily configured during the initial stages of the project, and that developers may not be aware of its presence in the solution (Fig. 2).

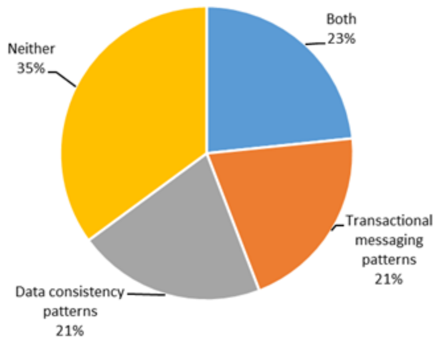


Figure 4. Data consistency and transactional messaging patterns usage in participants' projects

Maintaining data consistency is one of the most crucial challenges that can occur in distributed systems such as MSA. The additional difficulty is also to provide transactions in NoSQL databases that will work side by side with relational databases [30]. Also, eventual consistency - stabilisation of the system after distributed transaction, may cause problems with availability and scalability [24]. Due to these problems, design patterns maintaining data consistency should be introduced. Domain events are used in the Domain-Driven designed systems. They are published when the data is updated and can be consumed by other services. Domain events are often combined with aggregates (Aggregate pattern) that are modelled around one transaction in the system [31]. Aggregates emit domain events when they are created, updated, or deleted. When the process cannot be handled by one single microservice then the Saga pattern is used. Provides a mechanism that ensures the consistency of data between multiple microservices. One of the challenges related to Saga patterns is that they only provide ACD (Atomicity, Consistency, Durability), but without the isolation property [9]. The following pattern that can be used to maintain data consistency is Event Sourcing, in which changes in the application state are stored as sequences of state-changing operations [32]. In Domain-Driven Design (DDD) systems, this pattern can be easily adapted to store the changes of the aggregates, which may give the following benefits:

- the domain events published reliably,
- the history of the aggregates kept,
- facilitated combining of relational and object approaches,
- possibility to be combined with Saga pattern,
- providing access to "time machine" - travelling in history using changes between objects.

From the other side Event Sourcing might be inconvenient due to:

- steep learning curve,
- messaging-based approach which may result in higher complexity,
- evolving and deleting of data more complex than in traditional persistence,
- querying the event store is challenging.

A more detailed description of the advantages and disadvantages of Event Sourcing can be found in [9].

The use of patterns to maintain data consistency by the survey participants' projects is similar to transactional messaging patterns - 44% of respondents (34 participants), but the answer was marked by other participants. After combining the two results, transactional messaging patterns alone are used by 21%, data consistency patterns alone are used also by 21%, 23% are using both and 35% are not using either transactional messaging or data consistency patterns (Fig. 4).

The next questions were only available to those users who answered yes in the question about patterns usage to maintain data consistency. The most common approach for this is to use domain events (59% out of 34 responses). Event Sourcing pattern is used in 44% of the projects of the users of data consistency patterns. Aggregates are used in 38% of the projects, and Saga patterns are used in 32%. Other answers (2 out of 34 answers) are: 'Outbox' and 'Real models with consistency check run by serverless code'. 'Outbox' answer written by the participant is probably referring to the transactional outbox pattern described in the previous section. The results are visualized in Fig. 5.

Observability and monitoring. Each application must provide its Service Level Agreement (SLA), which is the contract between the company and their customers and set forth the expected service parameters [33]. To measure the overall MSA parameters and provide Quality of Service (QoS) metrics, observability patterns were introduced. Observability is often defined as a combination of metrics, logging and tracing [34]. The patterns that can be used for microservice observability are the following:

- Health Check API - exposes the endpoint which gives the information about health of the service often represented as state,

Figure 5. Data consistency patterns used in participants' projects

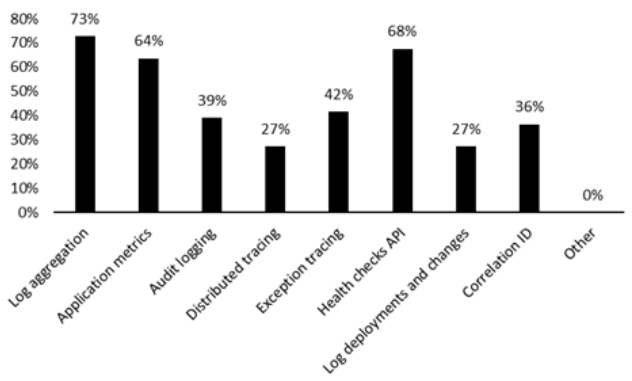


Figure 6. Service observability patterns used in participants' projects

- Log aggregation - centralized logging server which aggregates the information from log service activity and write logs and can provide alerting and searching functionalities,
- Distributed tracking - tracking the flow of the requests between services by assigning each external request an unique ID,
- Exception tracking - each exception is reported to exception tracking service which is de-duplicating an exception, alerts developers and tracks the resolution,

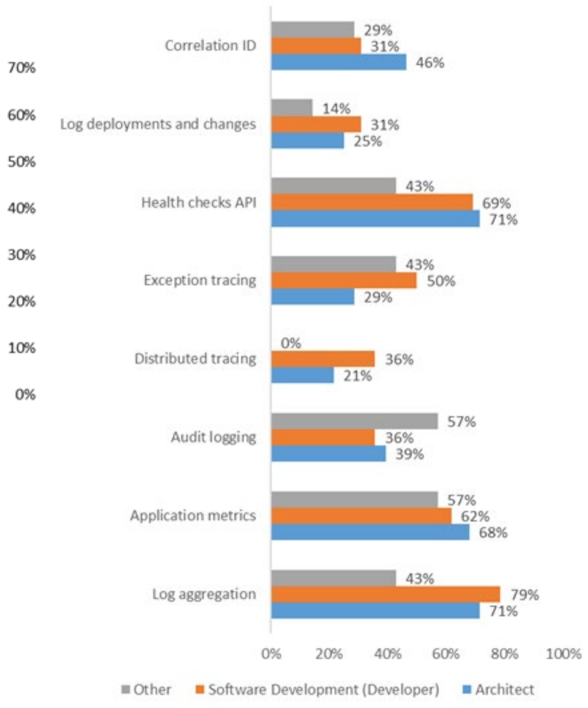


Figure 7. Observability and monitoring patterns usage divided by role in the team among all participants

- Applications metrics - metric server is aggregates the metrics maintained by microservices, such as counter and gauges, and prepare the visualization and alerts,
- Audit logging - records user actions in a database or file and enables searching, ensures compliance and detection of suspicious behaviour,
- Correlation ID - is similar to the distributed tracking, but is also used in queuing and in Saga pattern implementations. [23], [35].

The patterns mentioned above are described in detail in [9].

The survey participants largely declare that they use the service observability patterns (71% in total). In the following question the number of users rise (all the participants could mark one of the patterns) to 92% (only six participants did not mark any of the patterns). The most popular patterns are: log aggregation (73%), health check API (68%) and application metrics (64%). In around half of the projects these patterns are used: exception tracking (42%), audit logging (39%) and correlation ID (36%). Less popular service observability patterns, but still used in one quarter of the projects, are: distributed tracking (27%) and log deployments and changes (27%). The participants declare that in 8% of the projects there are no observability patterns used at all (6 responses without any pattern marked). There were no other patterns mentioned in the answers. The results are visualized in Fig. 6.

An analysis by team role across all participants shows that the most commonly used patterns (Health check API, Application metrics and Log aggregation) are similarly popular among architects, developers, and other roles (Fig. 7).

V. Discussion

During the survey, participants were asked about the design patterns that are used in their projects. Patterns are commonly used to solve recurring types of problems in software architecture [17]. The patterns analysed in the survey can be divided into three main categories: communication patterns, data patterns, and observability patterns.

Communication and reliability. The first two patterns analysed in the survey were API Gateway and Circuit Breaker. The API Gateway pattern is declared to be commonly used by the survey's participants. Implementation of this pattern gives the possibility to expose only a single layer of communication outside and hides the Microservices in the internal network. In contrast, Circuit Breaker is not as popular among participants (only twice as few as API gateway users). As a result, this can weaken the resiliency of the microservices architecture by impairing fault detection and leaving the system vulnerable to cascading failures [26].

The next set of patterns focused on querying the API topic. Participants declare that the CQRS (Command Query Responsibility Segregation) pattern is used more often than the API Composition pattern. CQRS separates read from write operations by querying different databases and keeping them in sync when any changes occur. One of the main benefits of CQRS is its clear separation of responsibilities between commands and queries, which contributes to cleaner, more straightforward, and easier-to-test code. It is a common practice to implement CQRS alongside API Composition, as combining these patterns can enhance system scalability and maintainability by clearly separating read and write concerns while efficiently aggregating data from multiple services. It is very surprising that only one third of the participants declared the usage of API Composition, but two third declared the usage of API gateway (which is one of possible implementations of API Composition). This may suggest a lack of understanding of this pattern among participants.

Service discovery. The next set of patterns focused on the discovery mechanism. Service discovery is used only in less than half of the survey participants' projects. Without this mechanism, the registration must be done manually, which raises the complexity of the final solution. On the other hand, huge complexity of the mechanism implemented from the beginning may lead to problems with deployment of the final solution. The service discovery can be implemented on either the client-side or the server-side. The server-side approach is far more popular among participants. The disadvantages of the server side are: maintenance of the router (if it is not cloud-based), problematic support of multiple protocols. It also generates more hops in the network compared to the client side. The last part of the service discovery is the registration mechanism. The most popular approach among participants is self-registration (each instance has the logic of how to register in a router), which gives them more control over the process.

Transactional messaging. Distributed transaction handling is the problem that is solved by the next group of patterns. These patterns were introduced to overcome the problems with distributed databases (database per microservice) and provide ACID transactions in the Microservices. Only less than half of the participants declare the use of transactional messaging patterns. This may lead to the conclusion that other patterns (e.g. data patterns) may be in use instead.

Maintaining data consistency patterns. The usage of data consistency patterns can be either an alternative or an extension for transactional patterns. Almost half of the survey's participants declared the usage of these patterns. The domain event pattern usage is declared by almost two-thirds of the participants, which may suggest the usage of Domain-Driven Design (DDD) in their projects. The aggregates are also defined in the DDD, but are used by only two thirds of the domain event users. This may lead to problems with the proper decomposition of MSA and maintaining the boundaries of microservices in the future.

Event Sourcing is implemented in less than half of the projects that use data consistency patterns. This pattern has a great benefit of storing the complete story of data changes in the whole system, but it also comes with higher complexity and problems with missing events. Saga pattern is designed to be an alternative for distributed transaction. The implementation of this pattern is highly simplified by dedicated libraries, which expose easy-to-use API and are often free to use. The Saga pattern is declared to be used only in one-third of the data consistency patterns users. The missing implementation of the Saga pattern is not very severe because it can be replaced with, e.g. the outbox pattern, but this pattern also gives the possibility to compensate (revert) the changes and orchestrate the processes. For other patterns, compensation and orchestration must be additionally implemented.

Observability and monitoring. The observability patterns are must-have in modern applications, which can also be found in the results of this survey. The large group (more than two thirds of the participants) declares the usage of these patterns. Log aggregation is declared to be the most popular pattern among the participants but is also often combined with the Health Check API. Health checks provide a quick way to detect when recovery mechanisms need to be triggered, while log aggregation allows for in-depth analysis of the issue and supports implementing improvements to prevent future occurrences. Application metrics are also a widely adopted pattern among participants and are essential for establishing a reliable Service Level Agreement (SLA) with future users.

It is quite surprising that design patterns in general are not as frequently applied in practitioners' projects as we could expect. Thus, it is advisable to design and implement a decision model to support MSA architects in the effective application of these patterns.

Comparison with the other MSA survey. When comparing the results of our survey with the other MSA survey ([17]), we can state that a similar set of design patterns is described as commonly used in Microservices. In [17], the authors used the Likert scale to describe the use of patterns, while in our survey, we simplified the answers to yes/no. In both surveys, the results are comparable; the most popular pattern is the API gateway. Sagas and Circuit breakers are used by one-third of the participants. In our survey, we can find the increase in the use of the CQRS pattern compared to [17]. In that work CQRS usage was described as "sometimes and less", when in our research almost half of the participants declare to use it. In our survey, we also explored the observability patterns (e.g., health checks, exception tracking, correlation ID) and extended patterns found in [17] with application metrics and correlation ID. The usage of application metrics gives the possibility to calculate Quality of Service (QoS) metrics, and correlation ID improves the tracking of messages in the system. Both of those mechanisms are used in the participants' projects. In addition, health checks are declared to be more commonly used in MSA than in [17]. We can find in our results the decrease in the usage of exceptions, which was the second most used pattern in [17]. Throwing of the exceptions is computational consuming and patterns like the Result pattern were introduced to overcome this drawback.

VI. Conclusions

Microservices-based architecture (MSA) provides great flexibility and scalability, making it an excellent choice for most modern, dynamic applications and systems. However, if not designed or implemented correctly, MSA can lead to significant performance bottlenecks, data consistency issues, and security vulnerabilities, among others. Thus, to fully harness the potential of MSA, architects must adhere to patterns that provide guidance on designing, implementing, and managing microservice-based systems effectively. MSA patterns cover a wide range of areas, including service decomposition, communication, resilience, observability, security, consistency, and more.

This paper presents a survey and its findings that illustrate how architects and the IT community nowadays practically engage with MSA, its paradigms, and patterns. It provides an overview of the patterns and techniques defined for and commonly used with MSA. The survey results indicate that architects and developers express a strong demand for patterns that ensure the reliability and security of the system.

The survey results show that the most commonly used pattern in microservices architecture (MSA) design is the API gateway, implemented in 73% of participants' projects. This pattern improves security by providing a single point of exposure to the public network. In contrast, a majority of respondents (62%) indicated that they do not employ service discovery patterns. While these patterns can be complex to implement independently, their adoption may be facilitated by the availability of established libraries and platforms. This omission can reduce the reliability of the system, as new instances must be added manually, increasing the overall complexity of the solution.

The general findings presented may offer valuable insight to architects and developers, highlighting which design patterns are beneficial to adopt in MSA projects.

Acknowledgments

We sincerely thank all the Participants who responded to the survey and contributed to the creation of this article.

Data availability

Data will be made available on request.

References

[1] J. Lewis and M. Fowler, “Microservices - a definition of this new architectural term,” https://martinfowler.com/articles/microservices.html, 2014, accessed: 2025-01-17.

[2] Z. Li, C. Shang, J. Wu, and Y. Li, “Microservice extraction based on knowledge graph from monolithic applications,” Information and Software Technology, vol. 150, p. 106992, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584922001240

[3] S. Newman, Building Microservices. O’Reilly Media, 2021. [Online]. Available: https://books.google.pl/books?id=aPM5EAAAQBAJ

[4] S. Li, H. Zhang, Z. Jia, C. Zhong, C. Zhang, Z. Shan, J. Shen, and M. A. Babar, “Understanding and addressing quality attributes of microservices architecture: A systematic literature review,” Information and Software Technology, vol. 131, p. 106449, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584920301993

[5] L. Qian, J. Li, X. He, R. Gu, J. Shao, and Y. Lu, “Microservice extraction using graph deep clustering based on dual view fusion,” Information and Software Technology, vol. 158, p. 107171, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584923000253

[6] V. Lenarduzzi, F. Lomio, N. Saarimaki, and D. Taibi, “Does migrating a monolithic system to microservices decrease the technical debt?” Journal of Systems and Software, vol. 169, p. 110710, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121220301539

[7] S. S. de Toledo, A. Martini, and D. I. Sjøberg, “Identifying architectural technical debt, principal, and interest in microservices: A multiple-case study,” Journal of Systems and Software, vol. 177, p. 110968, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121221000650

[8] S. Newman, Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith. O’Reilly Media, Incorporated, 2019. [Online]. Available: https://books.google.pl/books?id=iul3wQEACAAJ

[9] C. Richardson, Microservices Patterns: With examples in Java. Manning, 2018. [Online]. Available: https://books.google.pl/books?id=UeK1swEACAAJ

[10] W. Meijer, C. Trubiani, and A. Aleti, “Experimental evaluation of architectural software performance design patterns in microservices,” Journal of Systems and Software, vol. 218, p. 112183, 2024. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121224002279

[11] G. Vale, F. F. Correia, E. M. Guerra, T. de Oliveira Rosa, J. Fritzsch, and J. Bogner, “Designing microservice systems using patterns: An empirical study on quality trade-offs,” in 2022 IEEE 19th International Conference on Software Architecture (ICSA), March 2022, pp. 69–79.

[12] X. Zhou, S. Li, L. Cao, H. Zhang, Z. Jia, C. Zhong, Z. Shan, and M. A. Babar, “Revisiting the practices and pains of microservice architecture in reality: An industrial inquiry,” Journal of Systems and Software, vol. 195, p. 111521, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121222001972

[13] S. Henning and W. Hasselbring, “Benchmarking scalability of stream processing frameworks deployed as microservices in the cloud,” Journal of Systems and Software, vol. 208, p. 111879, 2024. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121223002741

[14] A. Hannousse and S. Yahiouche, “Securing microservices and microservice architectures: A systematic mapping study,” Computer Science Review, vol. 41, p. 100415, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1574013721000551

[15] M. Matias, E. Ferreira, N. Mateus-Coelho, and L. Ferreira, “Enhancing effectiveness and security in microservices architecture,” Procedia Computer Science, vol. 239, pp. 2260–2269, 2024, cENTERIS – International Conference on ENTERprise Information Systems / ProjMAN - International Conference on Project MANagement/ HCist - International Conference on Health and Social Care Information Systems and Technologies 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877050924016612

[16] A. S. Abdelfattah, T. Cerny, M. S. H. Chy, M. A. Uddin, S. Perry, C. Brown, L. Goodrich, M. Hurtado, M. Hassan, Y. Cai, and R. Kazman, “Multivocal study on microservice dependencies,” Journal of Systems and Software, vol. 222, p. 112334, 2025. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121225000020

[17] M. Waseem, P. Liang, M. Shahin, A. Di Salle, and G. Marquez, “Design, monitoring, and testing of microservices systems: The practitioners’ perspective,” Journal of Systems and Software, vol. 182, p. 111061, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121221001588

[18] R. Pinciroli, A. Aleti, and C. Trubiani, “Performance modeling and analysis of design patterns for microservice systems,” in 2023 IEEE 20th International Conference on Software Architecture (ICSA), 2023, pp. 35–46.

[19] Microsoft, “Cloud design patterns,” https://learn.microsoft.com/en-us/azure/architecture/patterns/, 2025, accessed: 2025-05-10.

[20] H. Farsi, D. Allaki, A. En-nouaary, and M. Dahchour, “Dealing with anti-patterns when migrating from monoliths to microservices: Challenges and research directions,” in 2023 IEEE 6th International Conference on Cloud Computing and Artificial Intelligence: Technologies and Applications (CloudTech), Nov 2023, pp. 1–8.

[21] G. Parker, S. Kim, A. A. Maruf, T. Cerny, K. Frajtak, P. Tisnovsky, and D. Taibi, “Visualizing anti-patterns in microservices at runtime: A systematic mapping study,” IEEE Access, vol. 11, pp. 4434–4442, 2023.

[22] S. Pulnil and T. Senivongse, “A microservices quality model based on microservices anti-patterns,” in 2022 19th International Joint Conference on Computer Science and Software Engineering (JCSSE), June 2022, pp. 1–6.

[23] I. Karabey Aksakalli, T. Celik, A. B. Can, and B. Tekinerdogan, “Deployment and communication patterns in microservice architectures: A systematic literature review,” Journal of Systems and Software, vol. 180, p. 111014, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121221001114

[24] X. Zhao and P. Haller, “Replicated data types that unify eventual consistency and observable atomic consistency,” Journal of Logical and Algebraic Methods in Programming, vol. 114, p. 100561, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2352220820300468

[25] X. Zuo, Y. Su, Q. Wang, and Y. Xie, “An api gateway design strategy optimized for persistence and coupling,” Advances in Engineering Software, vol. 148, p. 102878, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0965997820304452

[26] C. Lira, E. Batista, F. C. Delicato, and C. Prazeres, “Architecture for iot applications based on reactive microservices: A performance evaluation,” Future Generation Computer Systems, vol. 145, pp. 223–238, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X23001036

[27] C. Richardson, “Pattern: Server-side service discovery,” https://microservices.io/patterns/server-side-discovery.html, 2024, accessed: 2024-12-06.

[28] C. Richardson, “Pattern: Self registration,” https://microservices.io/patterns/self-registration.html, 2024, accessed: 2024-12-07.

[29] C. Richardson, “Pattern: 3rd party registration,” https://microservices.io/patterns/3rd-party-registration.html, 2024, accessed: 2024-12-07.

[30] M. T. Gonzalez-Aparicio, M. Younas, J. Tuya, and R. Casado, “A transaction platform for microservices-based big data systems,” Simulation Modelling Practice and Theory, vol. 123, p. 102709, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1569190X22001782

[31] V. Vernon, Domain-driven Design Distilled. Addison-Wesley, 2016. [Online]. Available: https://books.google.pl/books?id=h0u7jwEACAAJ

[32] S. Lima, J. Correia, F. Araujo, and J. Cardoso, “Improving observability in event sourcing systems,” Journal of Systems and Software, vol. 181, p. 111015, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121221001126

[33] O. Ghandour, S. El Kafhali, and M. Hanini, “Adaptive workload management in cloud computing for service level agreements compliance and resource optimization,” Computers and Electrical Engineering, vol. 120, p. 109712, 2024. [Online]. Available:https://www.sciencedirect.com/science/article/pii/S0045790624006396

[34] J. Kosińska, B. Baliś, M. Konieczny, M. Malawski, and S. Zielińnski,“Toward the observability of cloud-native applications: The overview of the state-of-the-art,” IEEE Access, vol. 11, pp. 73 036–73 052, 2023.

[35] S. Janapati, “Distributed logging architecture for microservices,”https://dzone.com/articles/distributed-logging-architecture-for-microservices, 2017, accessed: 2024-12-27