Journal of Edge Computing, **2023**, 2(1), pp. 84-101 https://doi.org/10.55056/jec.586

Reliable distributed systems: review of modern approaches

Oleh V. Talaver[1], *Tetiana A.* Vakaliuk[1,2,3,4]

[1]*Zhytomyr Polytechnic State University, 103 Chudnivsyka Str., Zhytomyr, 10005, Ukraine*
[2]*Institute for Digitalisation of Education of the NAES of Ukraine, 9 M. Berlynskoho Str., Kyiv, 04060, Ukraine*
[3]*Kryvyi Rih State Pedagogical University, 54 Gagarin Ave., Kryvyi Rih, 50086, Ukraine*
[4]*Academy of Cognitive and Natural Sciences, Ukrainian Branch, 54 Gagarin Ave., Kryvyi Rih, 50086, Ukraine*

Abstract. Called on to refine service-oriented architecture, a new architectural pattern named microservices emerged in the early two-thousands promising to speed up the delivery of new features. All of this is achieved by allowing many teams to work on separate autonomous parts of an application independently, though steepening the learning curve and introducing infrastructure-related issues. Overtime, while gaining more popularity, the initial meaning, as well as peculiarities of distributed systems development and reason for the use of the pattern, has turned out to be less prominent, instead becoming a sliver bullet for monolithic applications, which irreversibly leads to a tremendous increase in complexity of a system as well as other inherent problems. The absence of a single definition as well as the misleading name of the microservices pattern has all contributed to the development of the notoriously known distributed monolith. In this document, a review was conducted to resurrect the notion of microservices, understand their nature, and how and when the pattern should be applied. This is aimed to provide a reader that is yet not familiar with the peculiarities of distributed systems with enough information to make an informed decision of whether the pattern suits the business domain and problem at hand by describing structural approaches to modeling systems while stating the absence of focus on cross-cutting concerns like observability, deployment, testing. This is done by firstly understanding the difference between monolithic and distributed systems including their strengths and weaknesses, and defining the single most important reason for microservices to be used. The main part focuses on reducing the coupling in the system, which is the main obstacle during the development of a reliable and agile distributed system. Firstly, by modeling boundaries following domain teams and architectural needs, we ensure autonomous deployable units are created that provide independence during development, isolation, and reliability. Later by analyzing approaches to sharing data and communication, which are two major forces structuring a distributed system, we secure the previously established boundaries.

Keywords: microservices, distributed system, bounded context, coupling, domain-architecture-teams relation, design patterns, communication, data sharing

1. Introduction

In recent years, distributed architectures such as microservices (MSs) have gained much attention and popularity due to the possibilities the architectural pattern opens in terms of optimization, tech stack diversification, and such [7]. Although it's often presented as an architecture that is

olegtalaver@gmail.com (O. V. Talaver); tetianavakaliuk@gmail.com (T. A. Vakaliuk)
http://acnsci.org/vakaliuk/ (T. A. Vakaliuk)
0000-0002-6752-2175 (O. V. Talaver); 0000-0001-6825-4697 (T. A. Vakaliuk)

superior to a monolithic approach, it must not be seen as a replacement nor must it be followed blindly, as these architectural styles serve different goals and pose their problems.

When compared to distributed systems (DSs), monoliths are much quicker to develop as there is no need to think about how different components are communicating, what data a particular part should own, and how to share the data over the network.

The absence of physical boundaries between modules in a monolith, although allows for quick iteration on new features, quickly leads to the overall structure being lost, introducing unneeded coupling, therefore making the system harder to extend and support, leading to the so-called "Big ball of mud".

On the other hand, DSs, if built the right way, simplify the development process when many teams are involved, reduce the complexity of change, or dependency of teams on each other, and speed up development. Other benefits like resilience and easiness of comprehension are secondary because outweighed by the additional concerns that are brought with the introduction of such systems, not to say that monolith can still be built to be scaled, more resilient, and structured in a way that is easier to understand. The added concerns present a tremendous amount of complexity to the system, while such a common advantage as technological heterogeneity only increases the efforts needed to support the code base [3, 25].

For this document, a comprehensive review of the MS structuring, data sharing and consistency, and communication patterns between DS's components is considered as being the main drivers behind a reliable, extensible DS implementation, while leaving security, configuration management, monitoring, and other cross-cutting concerns out of scope. Also, as writing software means providing a solution to a given problem, a domain-oriented approach is followed to show the relationship between business, architecture, and teams that influence the use of MSs, the structure of DS a system, and design patterns chosen during a use case implementation.

2. Literature overview

In the following section, we present a general review of currently available related literature sources regarding MS development and data consistency within distributed systems. For the sake of reviewing alternative views on the topic, a few books have been chosen. The selection was based on the relevance of the literature to our subject matter, the rating and credibility in the developer's community, and the date of publishing. Other cited sources were chosen by searching the phrases and/or their combinations "Microservice", "Distributed system", "Data consistency", "Reliability", "Fault tolerance", "Domain-driven development", "Data sharing", "Communication" in the Google Scholar search engine and then picking relevant ones based on citation rate, publishing date, availability of free version on archives, the abstract provided. To present a fresh view of the books [13, 18, 23], they were studied after the main part of the document had been finished. While all the presented sources except [13] have a much broader view, this paper focuses on gathering thoughts, useful practices, and possible issues related to the modeling of MS boundaries and communication.

Posing the motivation to use MS architecture, both books [18, 23] list some advantages and limitations of microservices and monoliths; [18] refers to MSs as a pattern that allows for aligning organization structure and the system architecture, the same time considering a modular monolith with its ease of development; [23] states "The main goal of MSA is to achieve easy maintenance, quick software development with frequent deployment, short development cycles, and continuous delivery", while little information is provided regarding the structured monolith and the team collaboration or influence on architecture. The topic of team organization is discussed. At the same time, [13] focuses on the handling of data in larger, enterprise projects that most often include a distributed system; and defines the key concepts in any software as being: reliability, scalability, and maintainability, further explaining each.

In [18] the authors present a detailed business-oriented view of microservices, including team collaboration for effective development and usage of bounded context (BC) as a way to structure MSs and explain the process of splitting a monolith, though [18] elaborates more on the concept of coupling and cohesion and also provides alternatives ways to decompose a system that isn't concerned with the business but data, tech stack and organization structure. Sources [18, 23] present communication patterns with synchronous and asynchronous communication styles further describing common technological solutions for implementation but [18] adds a pattern of asynchronous communication through common data as well as describes GraphQL technology and gives more context about managing breaking changes, explains the difference between events vs commands and asynchronous programming vs asynchronous communication style, draws attention to different events based on content (full, partial, ids) and exhaustively explains resiliency of distributed systems (same as [23]). In comparison, this document has no technology being reviewed, instead, a more sophisticated evaluation of request types and their difference in influence on the coupling is presented.

Atomicity is addressed in all the sources, though [23] suggests microservices composition approaches like orchestration and choreography to be disconnected from the topic and are reviewed separately with much attention to business flow modeling languages. Given the data-oriented focus, [13] concentrates on the challenges of implementation of ACID principles and more relaxed guarantees of BASE systems transactions in detail. Focus is put on the different anomalies and problems with concurrent operations that can occur, especially in a distributed system. Then, the author continues with a thorough review of the distribution systems' unreliable nature to later discuss how consistency and consensus algorithms are implemented. The part revolves around the notion of linearizability defined in the CAP theorem and various ways the balance between consistency and availability is struck in diverse types of storage to allow specific goals like scalability, consistency, failure tolerance, and order to be fulfilled. Based on the previous chapters the final part addresses the common implementation of data derivation using batch or stream processing. Other cross-cutting concerns like testing described in [18] and [23] are out of this document's scope.

Overall, this paper gives another elaborated view on the topic of microservices modeling presenting many of the issues and possible solutions.

3. Theoretical background

A DS has components running outside a single process or network and communicates by exchanging messages. While the term DS is intelligible, the notion of MS is vague at least. The micro part implies that it's somehow related to size, moreover, it means that our components must be small, or rather the number of APIs is kept small. Small isn't an objective measure, however. We can easily split our application into such services, each having a single API endpoint, but will end up with many services that have one or two business API endpoints and dozens of integration endpoints. Named differently, it's a – big ball of mud (BBM) [11].

It isn't to say that the BBM is not useful, the architecture is widely used, and, without exaggeration, the largest part of the internet is written in the pattern. The problem is with the easiness of development of such a system. When systems extension is uncontrolled, new features are mindlessly stuck on top of others, and many dependencies cross the components, it makes it hard to develop [11].

So, it's probably more useful to think about what MS architecture is not:

- It is not about the easiness of delivery. It's easier to deploy a monolith than 300 MSs and monitor one thing than dozens.
- It is not a 10-liner or a small component that can be easily rewritten or understood. We do not break to understand the application easily, it's rather a byproduct.
- It is not about the usage of a tech zoo with many languages, other new and shiny technologies, the absence of Extensible Markup Language (XML) Application Programming Interface (API), or the presence of Representational State Transfer (REST) principles. It only increases the complexity of both monitoring and covering cross-cutting concerns for different tech stacks. A container orchestrator will not solve domain problems, they have another purpose.
- Neither is it for performance reasons. We can have a monolith and still make it work okay if we scale the data layer. Also, given that we decouple a system, the network becomes the new bottleneck that hinders performance [20].
- Problem locality may seem to be easier when we have isolated MSs, but an investigation usually starts from a use case not functioning properly which may involve a handful of participants in-between, and reading multiple logs trying to correlate requests without much observability is a challenging task.

To summarize, we do not create MSs just to replace monolith. There is no need to decompose monolith without a reason. Moreover, during the initial stages of development, a monolith gives many advantages in the easiness of iteration and allows further to be decomposed without premature problems [11]. When a company grows, however, it starts to be hard to coordinate teams and resolve conflicts causing efficiency to decrease. It's when MSs come in handy allowing designated teams to separate autonomous components that can only communicate through an API, allowing anything within to be changed, delivered, and iterated upon not affecting other teams. But the benefits don't come without a conscious understanding of design patterns and smells as otherwise coupling can be easily introduced to a system. The coupling itself is the measure of the independence of different components in a system. High coupling makes it harder to introduce changes and extend the system. When parts depend on one another, it's not always possible to be sure that everything will work if a change is introduced. Also, it may require multiple modules to be deployed together, which is unwanted.

However, we can't remove coupling completely because the system will be useless, so it must be ensured minimal between the units. The more coordination, re-usage, and storage sharing are present, the higher is coupling. Even event consumption is a contract, but a loose one [22]. Without constraints, the coupling can be introduced to any system, will it be a monolithic one or MSs. DSs, however, are less prone to problems just because of the physical separation that makes it harder to cross the boundary and break the constraints.

4. Modeling microservice-based systems

4.1. Establishing boundaries

The most important thing is to take control over sprawling features in a big system by carefully choosing boundaries. In MSs, the boundary is the surface that introduces coupling, the MS usually communicates with other ones, so it's required to strike the balance between the infrastructure API, the one that is used by other services, and domain one, the one that a business cares about [12].

Identifying boundaries isn't a one-time process, especially at the start of system development. That's why the monolith is so crucial in the stages because it won't only allow starting up quickly, but also decompose the application once we understand how to split the whole beyond the pure suppositions and experimentation.

So, how to decompose? As we have already understood, the size isn't a good indicator. We must break our system apart using cohesion as the guideline, not granularity, and Domain Drive Development (DDD) may help us with that. In DDD there is a description of an especially important concept called Bounded context, which is a separate unit in an application that is defined by domain meaning [1, 6].

This concept gives us the right granularity for our boundaries in a DS. BC may be a single MS or a group of such because it's not the physical but logical unit that is, however, bigger than classes, modules, and MSs therefore easier and more useful to maintain.

BC doesn't imply the use of any specific technology; it operates on a higher level. We may extract functional parts (to increase stability, depending on the task, e.g. to remove the stain from otherwise slow service) and place it within the same context allowing it more coupling with other components of the context, like using RPC or even share the same DB, because withing BC we allow coupling in, we know that we will need to make changes, we know that only one team handles that, but changes mustn't propagate outside [16].

Applying to DSs, the goal is to make the unit as autonomous, available, and independent as possible. It'll be easier to locate an error if we know that a dedicated part of an application is responsible for some functionality. It'll be much easier to extend the system, knowing that there is no strong dependency between different contexts. A team must handle one or a combination of such units, removing dependency on other teams and increasing agility.

Identifying the boundary.

BC should be split depending on the functional relation, and behavior and span across all the layers from data to view (using the micro-frontend technique). That is, we do not split the context depending on technical requirements like the platform (mobile, desktop) or the type of process (long-running or not). This is essential to ensure the absence of collisions between different teams during the development process [6].

As a hint, domain language can be used to find separate contexts. For example, we may have the same user identity across the whole application, but in various parts, the user may be used for different purposes with different data. So, we may have user billing information for the billing context, user shipping for the shipping context, and something like a lead in the marketing context. The data in the contexts is usually different or used for different purposes [6].

The three-way relationship.

It is as important to form our teams the right way. According to Conway's law, organizations design systems that mirror their communication structure. Without getting our organization's structure in order, we will not be able to get rid of the chaos in the code [5].

Therefore, we may see a three-way relationship between domain, architecture, and teams. When organization structure changes under the discoveries, we need to ensure that we also update our boundaries in the code and team structures accordingly to prevent cross-context teams [6].

How domain influences BCs.

There are many heuristics related to the way of defining a good boundary of an MS, but the most important one is related to the domain.

The domain is what an organization deals with, the problem it tries to solve. And some subdomains emphasize the importance of a particular part of our domain [24]:

- Generic – the stuff companies do the same way, like auth functionality, billing... The subdomain does not give business value.
- Core – the subdomains that provide a competitive advantage. The complex part that we want to work on as much as possible. Involves a lot of experimentation. This is part of the domain that we do not want to quickly decompose into MSs.
- Supporting – support core ones and contain the functionality specific to the domain that is not crucial, but the app cannot work without.

The separation provides a clear map of what to concentrate on, where to use complex approaches to modeling, like DDD, and where to outsource or buy a technology, like in the case of generic authentication solutions. Design approaches like DDD have their benefits but they add much complexity as well. There is no point in DDD for a create, read, update, and delete (CRUD) functionality. When we introduce complexity, we only make the system harder to comprehend.

Identify subdomain.

There are a few hints to identify the importance of an application part, all related to complexity [24]:

- CRUD is simple, there is no need to make it more complex – simple.
- Simple rules of validation are also simple.
- Complex algorithms, however, need more dedication – complex.
- Business rules or so-called invariants are a hint that the part is complex.

Once complex parts are found, generic ones are much easier to identify by comparing relatively. Of course, the complexity is influenced by the domain itself, so a generic part of an application for one company is the core for another [24].

The complexity of application about subdomain.

Tuning the complexity of part of an application to match the business value is vital:

- When the significance of the part of an application is not that big because competitors can easily catch up to us, we may simplify.
- When we under-investing time and effort into developing the part of an application that stands out and is not that easy to catch up to, we would rather reconsider the opportunity.
- When all the competitors catch up, the feature becomes just the expected one, leading to a shift of focus.
- Simplify all except the core subdomain. The complexity is probably accidental.

Domain always changes. Once generic part may become the core one. In the case of Slack which initially was a game development company and only later pivoted because of the absence of avail and started selling the chat app developed and used the whole time [24].

How data influences BCs.

Sometimes we cannot make a feature work without some data, therefore independent units become quite chatty. The reason may be that we put it into the wrong BC, the wrong MS, so redraw the boundaries. Other data-related heuristics are:

- Consistency control – two processes must run one at a time or be part of the same service/context.
- Linearizability or reading the last write – synchronous calls introduce higher dependency, therefore boundaries must be reconsidered.
- Tolerance to eventual consistency – hints that the units may be split.

Interactions between BCs.

Interesting things happen not in BCs but between them. If something goes wrong in one BC, it must not cause other BCs to stop working, this way we achieve resilience in our system. But it is an ideal case. Most often we deal with various kinds of relations between the components.

Using a context map, we can show logical relations. Arrows show dependency and there are at least three kinds of relationships [10]:

- Conform – when we conform, we accept the language of the other context, accept the logical structure of data, entities, and how they relate to each other. A context can conform to multiple others if there are no collisions in domain language (e.g., multiple have notions like "User").
- Partner – both contexts, when they talk, should understand the messages, therefore language, and schema. These are coupled and introduce high collaboration between teams, the collaboration is sometimes useful, and we will speak about it later.
- When we choose not to introduce such a dependency, we create an anti-corruption layer (ACL). By and large, it's a separate component that translates one request to another. The layer may be bigger than the system we protect, but that's okay if we strive to protect our structural part from other parts, otherwise, we end up with too much coupling. The creation of the translation layer is useful for cases when we want to separate a core subdomain from the supporting or generic ones. Also, it is an effective way to decouple legacy monolithic applications (aka strangler pattern [26]) or integrate with third-party services.

In the figure 1 the upper part is the physical communication between services, and the bottom one is the context map which helps in finding relationships. The arrows on the upper image show the flow of messages and the domain language used during communication. The context map shows us the logical relation.

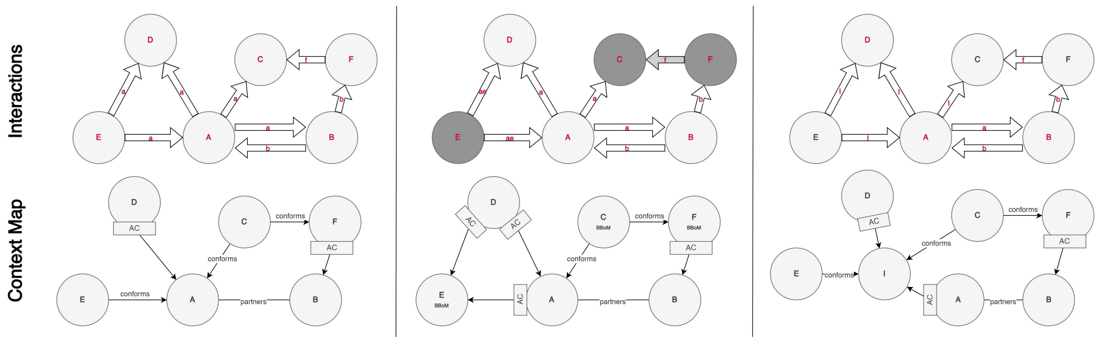

Figure 1: Difference between context map and BCs' physical interactions (first); error propagation in DS (second); creation proxy BC to allow language change (third) [6].

If we have an error in F and E contexts, it does not propagate far. As we can see in previous examples (figure 1), language A was used in many contexts, change in the language is impossible because it will break everything, so create interchange context with easier language than A. Like CDC only publishes changes, not domain events.

Logical boundaries within a monolith, are too permissive, so tend to break. The same goes for BCs, that is why we need to be conscious of the boundary.

Interactions between teams.

Interactions between teams are inevitable and unless coordinated consciously can halt the development process. There are possible types of team collaborations and how we can use them:

- Collaboration – teams work closely together.
- X as a Service – one team provides service for another to use.
- Facilitation – one helps another e.g. develops new features to the service they provided, makes necessary changes.

Once the core domain is discovered, validated, and has a clear roadmap, there is not much experimentation, and the solution must be developed quickly. X-as-a-Service removes the strong dependency and synchronization given by the close collaboration approach.

Otherwise, when we are exploring the core domain, the boundaries are not clear, and multiple teams work together and collaborate. The core domain should be broken into services much later because at this stage much is unknown, and it is required to iterate quickly to get the result. Moreover, one team may not have enough knowledge of the domain to finish the task, so teams must collaborate closely.

Teams are more flexible than application architecture, which can be employed to create temporary teams. Depending on the stage of development of a particular part of a system, collaboration approaches may transition from one another to facilitate experimentation or quick feature development.

Common modeling techniques are present to gain an understanding of the bigger picture of a system – Domain storytelling or refine a part by reconsidering boundaries – Event storming.

4.2. Sharing data

It would be useful to notice that the data is not the most important part of an application. Developing applications, we are trying to simulate real-world processes and data is a byproduct. Data, however, can unintentionally couple parts of our application and this section presents ways to consciously share data.

Data ownership.

A major characteristic related to data is ownership. As usual, there is no fast and hard rule, and everything is dependable on the domain. Once boundaries are established it is important to assign data to BCs and only make a more granular separation of data depending on the specifics of an application, functional, and domain requirements.

Again, we need to understand that we do not model data, we model behavior, so, for example, in an online shop system like Amazon, a shopping card may be modeled as a separate entity found in a separate context.

But the problem is that the shopping card concept just aggregates data from other contexts, creating a view or cache, which, among other things, couples the shopping cart context with all the other ones introducing too much complexity. What we want is to have each context represent the part of a shopping card (figure 2).

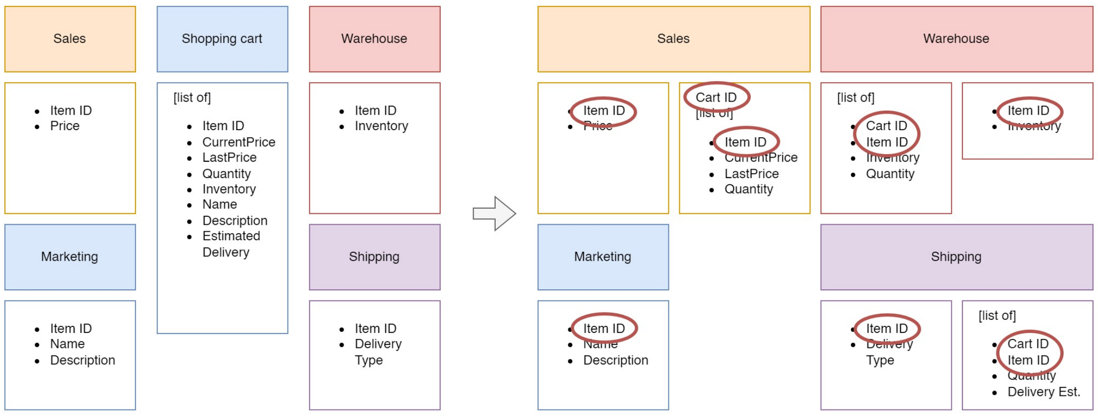

Figure 2: Shopping card as a separate entity (left) as a concept distributed among contexts (right).

The purpose of sharing is driven by the business.

The primary goal is to understand what the data is used for. It may be used to simply check something like whether the user has access and let him in. The other reason is to produce derivatives; if so, it may be an issue that we share data for other services to base their derivatives on, getting back to the relationships between different boundaries, remember to share consciously.

Some parts of a system need strict consistency (like the money transfer from one to another account) to ensure invariants while others may tolerate eventual one (e.g. user plan update propagation) to achieve better availability or distributed load (among workers) within context. As described in the CAP theorem, it's either one or another, we can't build a strongly consistent always available system. As a rule, when strong consistency guarantees are required, the parts should go into the same BC, we will talk about the reason for that in the next section. Most times it depends on the domain, as an example we need to ask experts whether it's okay if a user will be able to use a feature a minute or so after the time his subscription expires. In most cases, it's not a problem and eventual consistency is sufficient [8].

Approaches to data sharing.

To start with, there are multiple kinds of data: static, and mutable. Static is rarely updated and can be embedded in code (every or one service depending on whether we want to share it or have only one service to own it), live in shared DB, in configuration, or be extracted in other MS [2].

For mutable data, which is the most common and problematic one, there are the following approaches:

Reconsider talkative components.

Check if it is being extensively communicative with another service. We would rather redraw the boundaries and merge the two services.

Data aggregation gateway.

When we need to get data from diverse sources (MSs, BCs). Depending on the goal we want to achieve (analytics, display on front end) and the amount of coupling we can tolerate, there are at least those approaches:

API composition (aggregation) pattern – combine the data using a gateway. It isn't that suitable for situations that require joins over a big amount of data (for simple cases we can join in memory though). When the N+1 problem occurs, other approaches like CQRS are more suitable.

API decomposition pattern – a gateway splits data from a single request among multiple services and gathers a response. That is because each service needs only part of its data. atomicity must be insured for such cases because if we encounter an issue in later stages, however, we need to revert to all the earlier ones [2].

CQRS – Technique that separates read and write models to supply better read models for otherwise computationally intensive use cases. It is different from analytical storage (aka OLAP, Warehouse) in the sense of the purpose but is similar in that it is also a projection (cache, view).

Shared DB.

Use a single DB for multiple services. It is only advisable in a single BC because it is hard to track what must be changed across different MSs unless the DB schema is stored as a code and automatically checked, therefore, to simplify:

- Allow only one MS to write.
- Schema changes coordination must be present, so all the services must be within a single BC. Important to note that one team must deal with all dependent services.

The way data is stored may not satisfy all the use cases, in which situation other patterns must be used

Event sourcing.

This approach is like the way a source control system works (remember, however, that Git stores complete files, to the diffs), we use event store as the only source of truth. It is different from the event-driven system approach, where we save the state and only then publish the event. In event sourcing, we publish events and create a cache from the consumed events to speed up the process of querying. Given a small amount of data, all the events can be stored in memory.

The pattern should be used to achieve the following goals.

Business reasons:

- Auditing, compliance, transparency – the events stream provides the possibility to endlessly check it for patterns, and see individual steps that lead to a result of some kind.
- Data mining, analytics – as we have the stream with actions, we can analyze data starting from the beginning, not the time when the new requirements were defined.
- Deal with often changing requirements or unknown future requirements [17].

Technical reasons:

- Guaranteed completeness of raised events – once an event is published, it can't be deleted. It's only allowed to publish a new event to revert the changes made by the earlier one.
- Single source of truth – events store now contains all the information needed for a new service to be able to receive the needed information.
- Facilitates debugging – having a sequence of events, it's possible to look at them to see where an issue originated and what it caused.
- Replay into new read models (CQRS) – stream of events are easily convertible into a cache for different purposes.
- Deal with complexity in models – event sourcing forces us to follow narrow-focused models that are easier to understand.

Change data capture (CDC).

Updating caches synchronously makes the application less resilient and available. For example, if we update a search index or a Redis cache on each product update, what will happen if one of those components is not available? Will we skip the update or wait until the component responds? Not to say that we must never update external systems within the initial transaction due to the possibility of not recovering from failure; we will not be able to ensure atomicity in a DS, about which we will talk in the next section.

For cases like this, it would be convenient to use events for data update propagation. Events in case of events sourcing make the consumer understand them, and the approach is not as easy. In the case of CDC, we usually work only with dummy events of two types: upsert and delete, moreover, we do not opt into the only source of truth scenario by using techniques like DB transaction log tailing or other ones, therefore keeping the DB as the main source.

4.3. Communication

The next important bit apart from what to share is how to approach it. Communication design patterns handle how we build connections between components and are chosen based on many factors.

Strategy.

The communication strategy can be either synchronous or asynchronous, which is different from the protocol like HTTP or AMQP, it is rather a concept related to communication when we do not call other MSs within the initial request. We use events or other means to postpone these actions. “An HTTP-based approach is perfectly acceptable; the issue here is related to how you use it. If you use HTTP requests and responses just to interact with your MSs from client applications or API Gateways, that is fine. If you create long chains of synchronous HTTP calls across MSs, communicating across their boundaries as if the MSs were objects in a monolithic application, your application will eventually run into problems with performance, coupling, and failure propagation” [4].

Communication purpose.

The communication purpose refers to whether we want to query some data or mutate it and the purpose of the communication.

In case, we care about the condition being true later after the initial request, basically we want to reserve some resources and therefore strive to achieve atomicity in a DS. The same goes for a series of mutations that must succeed all or none.

Atomicity in a DS.

Atomicity is first driven by the domain. Do we need the operation to be atomic? Do we care about reverting changes when multiple services are called, and we fail in the middle while having data affected by those other services? Do we care about the reversion of changes when we update two different documents in the NoSQL database (DB) and fail before updating the second? We must always assume that if problems can happen, they will eventually happen, so will the business tolerate it?

There is a dependency between resource and the transaction-atomic boundary that is influenced by the way we tolerate the partial unavailability of a system (according to CAP theorem) and system type.

- Everything is transactional in SQL Databases that support ACID transactions; they scale poorly, however.
- A single document written is atomic in NoSQL databases that support BASE principles. And even if transactions are available in some databases like MongoDB, these perform poorly and are rather an exception than the norm. This, however, does not mean that it is not possible to atomically update multiple documents, it is just that a different approach is needed [9].
- There is no transactionality in disparate resources, such as a database and a queue. For this one, we use a pattern called outbox that preserves all published events in a single transaction with the DB write to be then handled and published [14].

So, when we talk about communication between different MSs, there is no atomicity unless we use one of the approaches: 2-phase commits or Saga pattern. These approaches are different depending on the communication strategy, where synchronous is used by the 2-phase commits and asynchronous by the saga pattern.

2-phase commits.

It is probably the first approach that comes to mind when operation atomicity must be insured across multiple MSs: let us create a long-lived transaction in each service we communicate with and update the data but only commit the transaction when we receive the second, confirmation request. Much can go wrong, especially if there are problems with the network. Moreover, the approach requires all the services to be up [15].

Saga pattern.

Compared to the earlier approach, instead of fitting everything into the same transaction boundary, we accept eventual consistency. The idea is to break the entire process into stages that are triggered or reverted using a message. In case of one stage failure, all others must be reverted by listening for and handling a compensating event [21].

When working with Sagas, observability is necessary due to the complexity of communication between many components. Another culprit is the need to implement and test compensating operations which is not that easy. Until a saga is committed, the data can be read dirty; in different contexts, there may be a need to read the dirty information as well, for example, in banking there usually is a clear and dirty account balance, where the foremost defined by all the completed transactions and the later one accounts those in the progress as well. Those problems can easily be solved by saving ongoing sagas in a different table and reverting completely or just using the data when needed.

There are two ways to implement Sagas: orchestration and choreography. It is not like one is superior to another, which is a quite common message, these both are used in different situations. The difference between them is the difference between commands and events, this will be explained a bit later, but in general, we decide whose responsibility it is to call, and how many components we talk to.

Consistency.

There are cases when we need the data to be precise and when strong consistency is needed, either the MSs must be combined into one or a synchronous call is used. As it has been said, however, in most cases, we tolerate eventual consistency for data – the user may have been downgraded. Still, it is not a problem if we allow him to use a feature until the plan change is propagated.

Service location.

Whether the participants are:

- Internal – our infrastructure, can use diverse technologies.
- External – external clients subscribe to receive events or use other means like webhooks, and RSS with sane timeouts, and exceptional care.

Request type.

Not all requests are alike, sometimes we just get data, sometimes mutate state, and sometimes just notify external listeners.

- Queries – Queries are requests to get some data when the client knows the destination service, and schema and receives the payload returned.

- Commands – Commands are requests to change something when the client knows the destination service, and schema and receives the payload returned.
- Events – Events show something that has happened when the publisher does not care about the consequences (what is triggered in other parts of an application and the results of it). They do not have a defined receiver, nor do they have a response payload. Despite being asynchronous, events do not take long to be handled unless there is a problem in a system; observability must be present to detect such situations. Events, however, are much more sophisticated because the consumer data and granularity of events are unknown upfront. If we create an event per single action, this may simply copy the processing logic from the event's producer to the consumer because the consumer must understand the order in which to handle the messages and what they mean. Also, the data that is used in the events can couple, especially if the event has more data that is required by a single consumer, trying to satisfy multiple.

What we need to understand first is whether services have the same reason, and rate of change, whether we can decouple even further, or maybe they must be one and prevent the chattiness. What changes together, goes together.

With MSs, we want to ensure an as minimal as possible interface. Therefore, we should think about what we want to share and create an ATL to prevent exposure of private, internal service knowledge.

By exposing domain events, we make the consumer understand and process the event, which introduces coupling. In case only one other service understands the language (one process stopped here, another starts there), we may allow it, but if there are a couple of such services, that may pose a problem.

For a consumer, it isn't always clear what events mean. Therefore, it's important to make them more intelligible, removing ambiguity.

Types of events.

To give more insights, there are different event types depending on the payload:

- Notification event (aka shallow event) – holds only IDs; the less data the event has the fewer efforts it's required to keep schema. The downside is the service publishing the event will be bombarded with calls for more data.
- Event-carried state transfer – holds a payload that may have different forms: only updated fields, delta. This type of event increases availability because the consumer receives all the required data contained in the event but also introduces schema maintenance problems and problems with the external cache consistency. Given the pros and cons, the proper way is to use the events in between BCs but use an ACL to map events to more specific ones with proper granularity and payload for different BCs.

Depending on the usage, events can be split into the following categories:

- Domain-specific – has fewer data and more meaning. Must be used within a single context and carefully propagated outside, with mapping to more specific ones.
- Integration (state changed) – synchronization purposes; pure payload without much meaning.

Order of events.

Some events can be handled out of specific order while for others ordering is crucial. Out-of-order messages aren't rare, given that multiple events producers or consumers are present, there is a possibility that either an event is published later or is consumed later than the action happened (compared to other actions) [19].

Situations like this can be mitigated by using partitions and assigning an event to a particular partition based on message data, like user ID, but it is hard to achieve in some systems like Rabbit MQ, where, if compared with Kafka, no native support for partitions present.

The rule of thumb is to develop a system so that it does not depend on message ordering. For example, publish the next event only after the earlier was handled and an acknowledgment was sent (in the Saga pattern) or consider cases when events are out of order. For example, an order can be canceled before it is accepted. Usually, it is hard to think through all the possibilities, so generative testing may be beneficial.

Relation between events and commands in terms of dependency.

Both events and commands introduce dependency. When a client sends a command, it knows the schema and must be changed when it does. Events have the schema as well, the difference is that now the publisher does not know anything about the consumer, therefore the dependency reverses, but coupling stays. In both cases, we need to track who consumes the API to know what to change. This is the same as the dependency inversion principle in the layered architecture, we use it to move the responsibility to the right side.

Idempotency for commands and events.

Due to how DSs work, it is not rare when a web service may receive two identical requests instead of one. This problem may happen when we make a synchronous request, but a network partition occurs on the response, so we retry. The same applies to asynchronous systems, when a message is published to a broker and then a network partition occurs, we retry publishing, or when we consume a message, and cannot acknowledge the broker (or in the broker itself if otherwise is not guaranteed).

The problem may be solved with the introduction of idempotency:

- Making upsert (update and insert), it does not matter how many times we run the operation.
- Save the ID of handled requests and check if it is not already handled (idempotency key).

Airbnb implemented "Orpheus", a general-purpose idempotency library, across multiple payment services with an idempotency key which is passed into the framework, being a single idempotent request.

5. Conclusions

In this article, an overview of approaches for modeling MS based systems was presented alongside technical challenges commonly encountered while embracing the architectural pattern. This is intended to equip a reader who is still unfamiliar with the nuances of the DS with sufficient knowledge to make an informed decision on whether the pattern aligns with their specific business sector and problem to solve. The focus lies on structural methods for system modeling, though it explicitly does not delve into cross cutting concerns such as observability, deployment, and testing. Designing a reliable DS isn't easy and must be approached consciously. The nature of DSs differs dramatically from what we used to think when dealing with monoliths and well-structured MS architecture is a leading and yet being studied approach used to build a reliable system as it does not imply the use of any specific technology and therefore provides the needed flexibility most importantly to organizations with many teams, which feels novel after its precursor – Service-oriented architecture that has a major dependency on the smart middleware called Enterprise service bus and communications between services which makes it a problem to create an easy to extend system. Though flexible, MSs bring the problem of keeping the structure loosely coupled to the table, which was the main topic of discussion in the document. MSs are just building blocks, and we must strive to put them together so that it's easy to change under the influence of functional requirements from the business, non-functional needs, and the structure of an organization. When MSs compared monoliths, even though both are built to satisfy the business need, the technical goals that are achieved vary a lot. So, for the MSs it's the agility that attracts enterprises to embrace the system, opposing a monolith that speeds up development but tends to lose structure, therefore being less and less maintainable. The crucial part to take out of this article is not to join the cause of MSs development unless for the learner's sake, attracted by the novelty of the shiny new piece of technology and the possibility of usage of a diverse set of tools and patterns like a container orchestrator, event sourcing, etc., but carefully evaluate the complexity to be brought.

As such, the biggest problem in all without-exception systems is the ease of extensibility and maintenance, which has been a major concern since the beginning of time. Various paradigms are developed to solve problems that follow a common pattern efficiently, MSs are one such way.

Developing MSs, we strive to minimize the dependency between units directly influenced by the organization's structure and business domain. While reducing dependency, we want to concentrate on the important parts of our domain, those that will bring the most value and competitive advantage, therefore we strive to protect those parts with boundaries. The business domain is the leading driver of the way we define the boundaries and communication between them and teams. While communicating, we share data, the prime contributor to the increase of coupling in a system, so we must be sensible when defining what we communicate. Another aspect is the communication itself, which defines the topologies and connections between the units in a system, common patterns, and approaches to the resolution of atomicity and awaking behavior in one part from the other must be noted.

A couple of relative topics are vital to developing complex DSs. Although BCs increase the problems' locality, we still need absolute observability in a DS. Another problem is dependencies. When there are many parts in a system, it's hard to track dependencies by using diagrams, we need to know what external components will be influenced and test them.

References

[1] Balalaie, A., Heydarnoori, A., Jamshidi, P., Tamburri, D.A. and Lynn, T., 2018. Microservices migration patterns. *Software: Practice and Experience*, 48(11), pp.2019–2042. Available

from: https://doi.org/10.1002/spe.2608.

[2] Bozan, K., Lyytinen, K. and Rose, G.M., 2020. How to Transition Incrementally to Microservice Architecture. *Commun. ACM*, 64(1), p.79–85. Available from: https://doi.org/10.1145/3378064.

[3] Bucchiarone, A., Dragoni, N., Dustdar, S., Larsen, S.T. and Mazzara, M., 2018. From Monolithic to Microservices: An Experience Report from the Banking Domain. *IEEE Software*, 35(3), pp.50–55. Available from: https://doi.org/10.1109/MS.2018.2141026.

[4] Communication in a microservice architecture, 2022. Available from: https://learn.microsoft.com/en-us/dotnet/architecture/microservices/architect-microservice-container-applications/communication-in-microservice-architecture.

[5] Conway, M.E., 1968. How do committees invent. 14(5), pp.28–31. Available from: https://www.melconway.com/Home/pdf/committees.pdf.

[6] Evans, E., 2003. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley Professional. Available from: https://sd.blackball.lv/library/domain-driven_design_-_tackling_complexity_in_the_heart_of_software.pdf.

[7] Francesco, P.D., Malavolta, I. and Lago, P., 2017. Research on Architecting Microservices: Trends, Focus, and Potential for Industrial Adoption. *2017 IEEE International Conference on Software Architecture (ICSA)*. pp.21–30. Available from: https://doi.org/10.1109/ICSA.2017.24.

[8] Gilbert, S. and Lynch, N., 2002. Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services. *SIGACT News*, 33(2), p.51–59. Available from: https://doi.org/10.1145/564585.564601.

[9] Helland, P., 2016. Life Beyond Distributed Transactions: An Apostate's Opinion. *Queue*, 14(5), p.69–98. Available from: https://doi.org/10.1145/3012426.3025012.

[10] Kapferer, S. and Zimmermann, O., 2020. Domain-specific Language and Tools for Strategic Domain-driven Design, Context Mapping and Bounded Context Modeling. *Proceedings of the 8th International Conference on Model-Driven Engineering and Software Development - MODELSWARD*. INSTICC, SciTePress, pp.299–306. Available from: https://doi.org/10.5220/0008910502990306.

[11] Khononov, V., 2018. Tackling Complexity in Microservices. Available from: https://vladikk.com/2018/02/28/microservices/.

[12] Khononov, V., 2020. Untangling Microservices, or Balancing Complexity in Distributed Systems. Available from: https://vladikk.com/2020/04/09/untangling-microservices/.

[13] Kleppmann, M., 2017. *Designing Data-Intensive Applications*. O'Reilly Media, Inc. Available from: https://www.oreilly.com/library/view/designing-data-intensive-applications/9781491903063/.

[14] Laigner, R., Kalinowski, M., Diniz, P., Barros, L., Cassino, C., Lemos, M., Arruda, D., Lifschitz, S. and Zhou, Y., 2020. From a Monolithic Big Data System to a Microservices Event-Driven Architecture. *2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. pp.213–220. Available from: https://doi.org/10.1109/SEAA51224.2020.00045.

[15] Lin, Q., Chang, P., Chen, G., Ooi, B.C., Tan, K.L. and Wang, Z., 2016. Towards a Non-2PC Transaction Management in Distributed Database Systems. *Proceedings of the 2016*

International Conference on Management of Data. New York, NY, USA: Association for Computing Machinery, SIGMOD '16, p.1659–1674. Available from: https://doi.org/10.1145/2882903.2882923.

[16] Merson, P. and Yoder, J., 2020. Modeling Microservices with DDD. *2020 IEEE International Conference on Software Architecture Companion (ICSA-C)*. pp.7–8. Available from: https://doi.org/10.1109/ICSA-C50368.2020.00010.

[17] Müller, M., 2016. *Enabling retroactive computing through event sourcing*. Universität Ulm. Available from: https://doi.org/10.18725/OPARU-4111.

[18] Newman, S., 2021. *Building Microservices*. 2nd ed. O'Reilly Media, Inc. Available from: https://www.oreilly.com/library/view/building-microservices/9781491950340/.

[19] Pritchett, D., 2008. BASE: An Acid Alternative: In Partitioned Databases, Trading Some Consistency for Availability Can Lead to Dramatic Improvements in Scalability. *Queue*, 6(3), p.48–55. Available from: https://doi.org/10.1145/1394127.1394128.

[20] Rotem-Gal-Oz, A., 2008. Fallacies of Distributed Computing Explained. *Doctor Dobbs Journal*. Available from: https://www.researchgate.net/publication/322500050_Fallacies_of_Distributed_Computing_Explained.

[21] Štefanko, M., Chaloupka, O. and Rossi, B., 2019. The Saga Pattern in a Reactive Microservices Environment. *Proceedings of the 14th International Conference on Software Technologies - ICSOFT*. INSTICC, SciTePress, pp.483–490. Available from: https://doi.org/10.5220/0007918704830490.

[22] Stevens, W.P., Myers, G.J. and Constantine, L.L., 1974. Structured Design. *IBM Syst. J.*, 13(2), p.115–139. Available from: https://doi.org/10.1147/sj.132.0115.

[23] Surianarayanan, C., Ganapathy, G. and Pethuru, R., 2019. *Essentials of Microservices Architecture: Paradigms, Applications, and Techniques*. 1st ed. Taylor & Francis. Available from: https://doi.org/10.1201/9780429329920.

[24] Vernon, V., 2016. *Domain-Driven Design Distilled*. Addison-Wesley Professional.

[25] Villamizar, M., Garcés, O., Castro, H., Verano, M., Salamanca, L., Casallas, R. and Gil, S., 2015. Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. *2015 10th Computing Colombian Conference (10CCC)*. pp.583–590. Available from: https://doi.org/10.1109/ColumbianCC.2015.7333476.

[26] Yoder, J.W. and Merson, P., 2022. Strangler Patterns. *Proceedings of the 27th Conference on Pattern Languages of Programs*. USA: The Hillside Group, PLoP '20, pp.1–25. Available from: https://dl.acm.org/doi/abs/10.5555/3511065.3511076.