If (domain logic) then (solution) - Distributed systems episode

Introduction

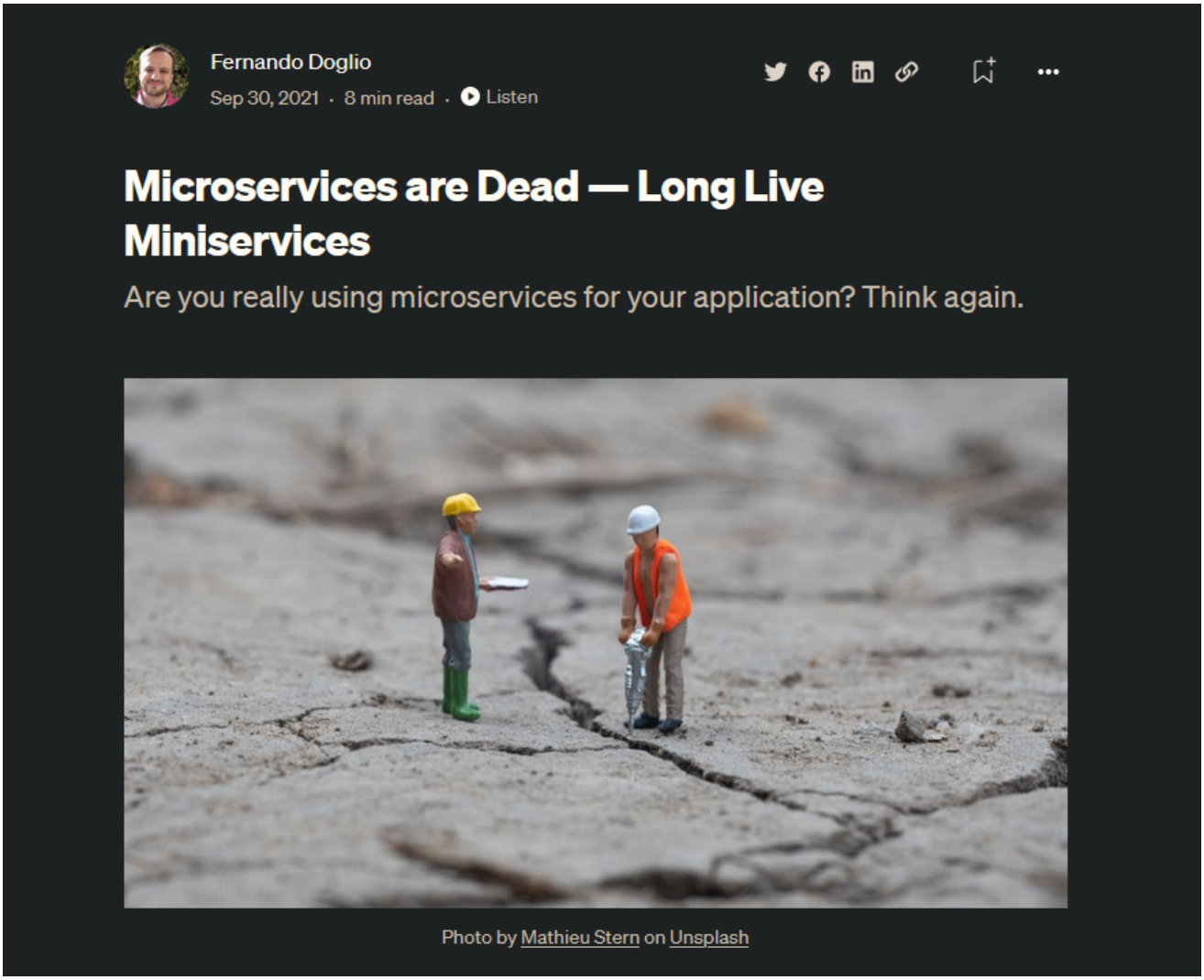
What if I say, that shared database pattern is useful, 2-phase commits aren't that harmful and HTTP isn't that dreadful?
We, programmers, like going from one *extreme* to another. Microservices are not a piece of cake; all of those methods are possible, you can use *{{pattern, approach name here}}*, moreover you have to use it, the only question *when*

Technologies come and go, principles, however, stay for decades. Like these, we're going to talk about.

I'm not going to tell about something new nor specifically about the fundamentals of distributed computing (for the last I suggest watching the video where the ground-up approach is explained). It's rather a practical meta-analysis based on the materials I have collected and my thoughts on different topics. The goal I wanted to achieve is to collect nuggets from different sources, systemize and research some beyond the boundary of superficial understanding.
Despite my perfectionism, it's not possible to process all the resources on the internet in a sane amount of time, therefore I'll be glad to hear your ideas, suggestions, and sources on the topic discussed.

What are microservices

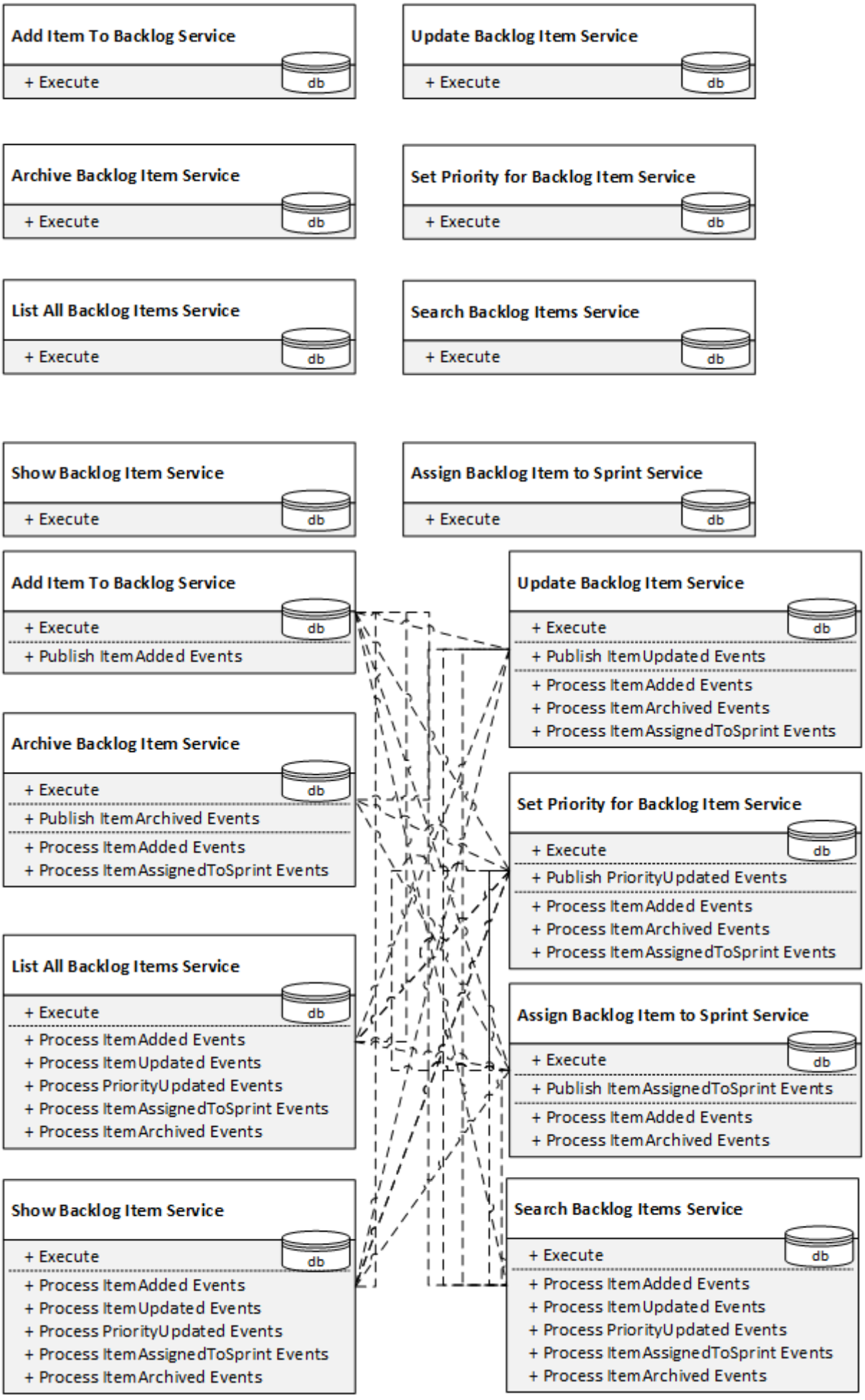
Over the years, the term *microservice* has become something like a buzzword. Everyone speaks about it, some use it because this is cool, and someone has already buried them but in the end, only a few understand what it is.



To prevent ourselves from falling victim to common cognitive bias, let's go down the rabbit hole to uncover the truth.

So, a distributed system is several components that are not running in a single process and communicate by exchanging messages, simple as it is. I'd argue that every system we develop is a distributed one just because we may use a database that is already out of process external component, not to speak about web applications that have UI running in a browser, communicating with the backend.

While the term *distributed system* is intelligible, the notion of *microservice* is vague at least. The *micro* part implies that it's somehow related to size, moreover, it means that our components have to be small, or rather the number of APIs is kept small. But again, *small* isn't an objective measure. We can easily split our application into such services, each having a single API endpoint, but we 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*. [1] [2]



[Image source]

I do not necessarily say that the muddy ball 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 terribly hard to develop. This causes the mud to sprawl with its tentacles, devouring any structure and turning it into spaghetti code. 1 3

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

- It is not about the easiness of delivery. It's easier to deploy a monolith than 300 microservices 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 definitely not about the usage of a tech zoo with many languages, other new and shiny technologies, the absence of XML API, or the presence of REST principles. It only increases the complexity of both monitoring and implementing the whole thing for different tech stacks. Docker and k8s will not solve domain problems, they have another purpose.
- Neither it is for performance reasons. We can have a monolith and still make it work ok if we scale the data layer. The Signal server is still a monolith server instance that handles many requests worldwide.
  To summarize, we do not create microservices to fix monolith Great monolith. So, there is something other to microservices that make them so widely used. 2 1 4

Image source

What we need to understand is that monoliths are great, we do not need to decompose them without a reason. It isn't the same as a *big ball of mud*. You have to start with a monolith to decompose the application without premature problems. 4

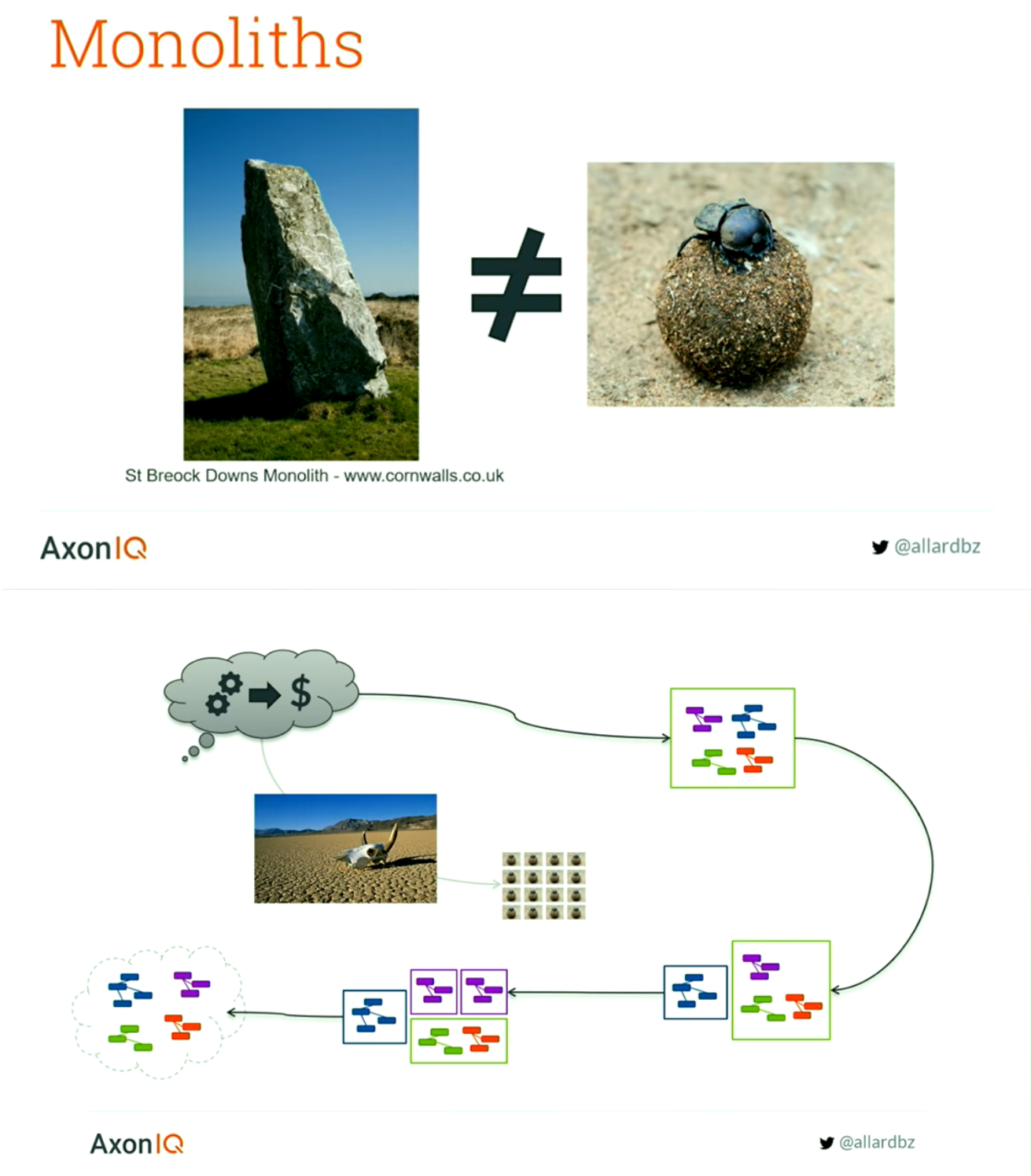


Image source

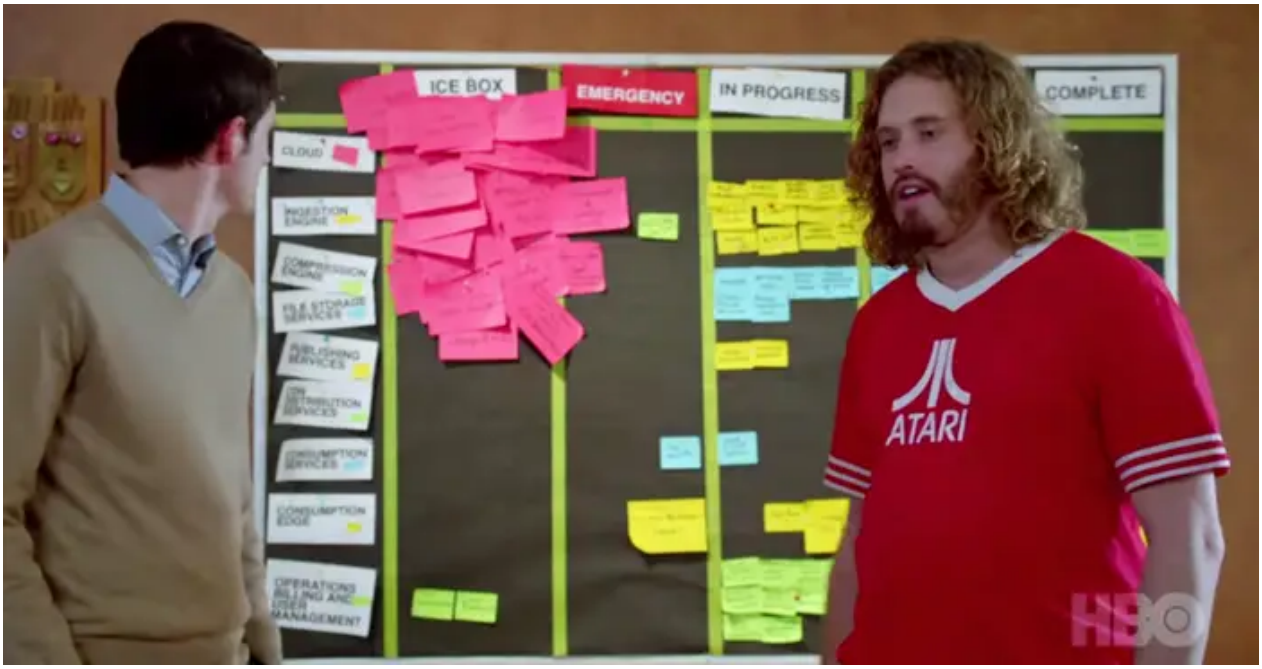
The single reason

Being Engineers, we have to deliver a solution to a business problem at hand, and this is what the name of the article is implying. So, if you could name a *single*

reason for microservices to be used, what it would be?

Agility

Microservices have a tremendous impact on agility, 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. [2]

> Joke, oh no, I meant Agility, skip the first slide
>
> 
>
>
> Image source: "Silicon valley" series

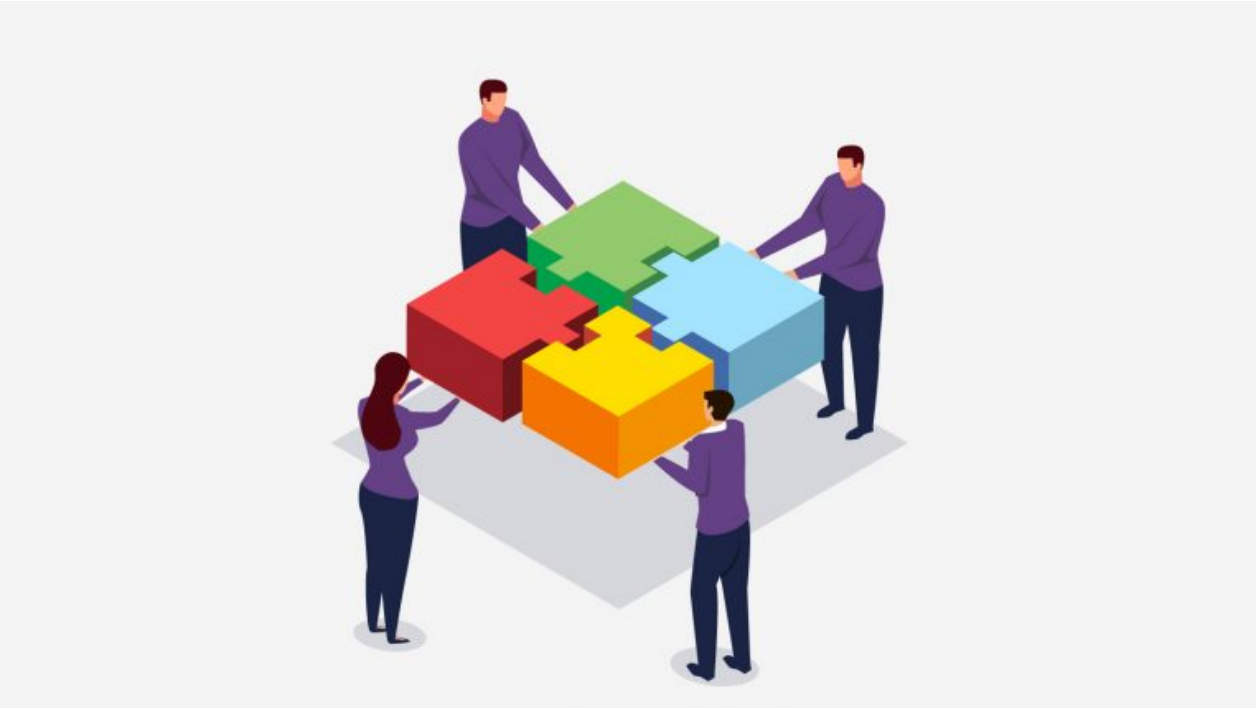


Image source

Again, if we do not need the above advantages, there is no reason to decompose, at least for now, because microservices introduce much complexity. When you have dozens of open pull requests in a repository developed in parallel, that's possibly a breaking point when you need to at least think about the change.

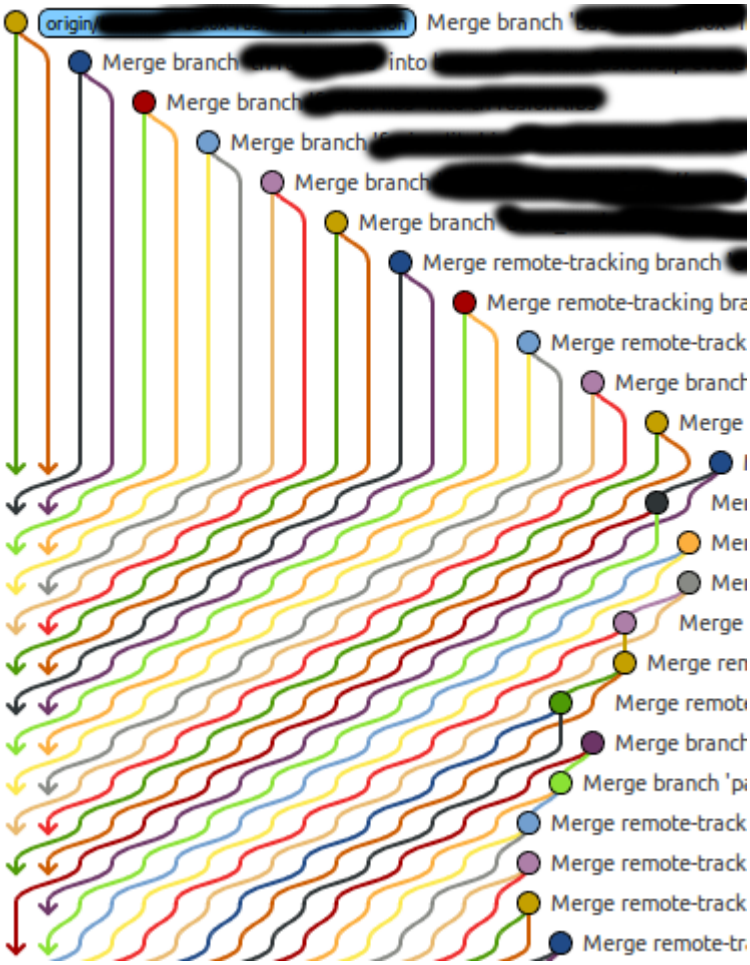


Image source

Overhead costs

Not pros alone. With great power comes great responsibility. Because of the complexity of microservices, there are additional concerns we need to ensure are covered:

- Loose coupling - without which microservices will turn into a distributed mess.
- Ensure easy problem locality - being able to identify the source of a problem; which service is responsible for producing wrong data. It refers to autonomy and ownership of some functionality by a unit of a system. It's very hard to identify where a problem originated without proper separation of concerns and monitoring.
- Easily see what changes have to be made, and dependent on the changed module parts (e.g. what microservices to update when we update this one)

Image source

The lurking problem

if there is a single point to remember from the article, it is that we only can achieve *agility* in the process of development if the coupling is minimal.

What is coupling? It 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, you can't always be sure that everything will work if a change is introduced. Also, it may require multiple modules to be deployed together, which is unwanted.
On the other hand, we can't remove coupling completely because the system will end up being useless, so it has to be ensured minimal between the units. The more coordination, re-usage, and storage sharing are present, the higher is coupling. Even events consumption is a contract, but a loose one. 2 5 6

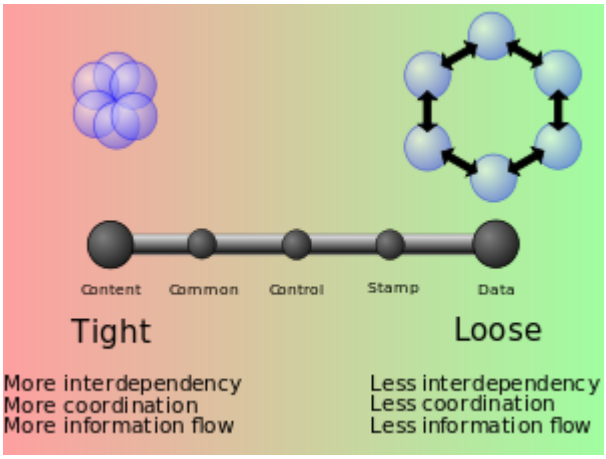


Image source

It's tough to understand abstract notions like coupling without understanding nature first. The coupling is caused by concentrating either on resolving local or global complexity too much, neglecting the other one. Local complexity is the complexity of each individual microservice, while global is related to the system overall. Paying too much attention to the local complexity, we end up with small, simple microservices that call each other without control. If we optimize the global one, we end up with a monolith that is hard to extend. 7

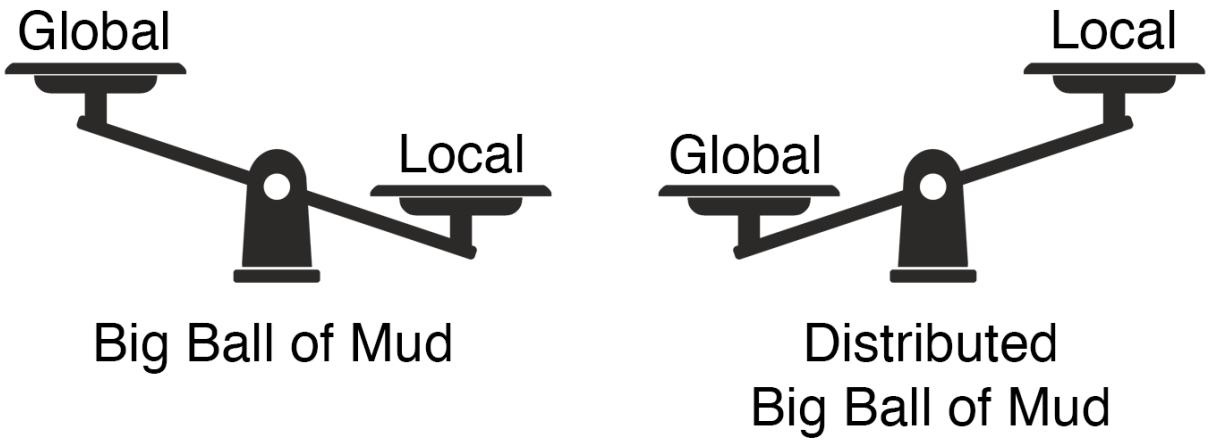


Image source

Coupling relation to the distributed systems

When talking about coupling the following classification is common, starting from the highest to the lowest: 5

- Code reuse (Content coupling) - you may reuse common utils library, but sometimes when duplications aren't common (in 2 or 3 places) it isn't necessary to extract the common part, as the overhead of future change may be higher. Also, consider when the common parts are owned by different teams or those parts can diverge because requirements aren't quite the same in all the places, therefore the parts do not change at once.
- Something shared externally (External coupling) - like protocol (e.g. IoT), variable, or shared DB.
- Controlling (Common coupling) - when passing arguments or sending commands, the controller knows about the worker and what to pass
- Data structure coupling (Stamp coupling) - you pass some data, a consumer of which has to know the structure, so when the structure is changed the consumer also has to be changed. In the rules engine, for example, individual rule

executers depend on the mediator, where these register, therefore change in structure causes all the connected components to be changed as well. Another example is the Saga orchestration approach, where the orchestrator is the only component that changes when the data structure changes because it is dependent on other actors of the Saga. But in both cases, we can change the components almost without constraints (the only constraint is the data passed)

- Data coupling - shared data dependency that is common in composition (e.g. rule engine), pipelines, and any other communication between components in a system. Prevention requires passing only the data required.

Overall, introducing excessive dependence between components in the system renders microservices useless. The system becomes a bunch of functions and classes separated by the network. We must never assume that inter-process calls to a function are the same thing as RPC.

Some of the Network fallacies

In a distributed system, we must never assume that a network is reliable.

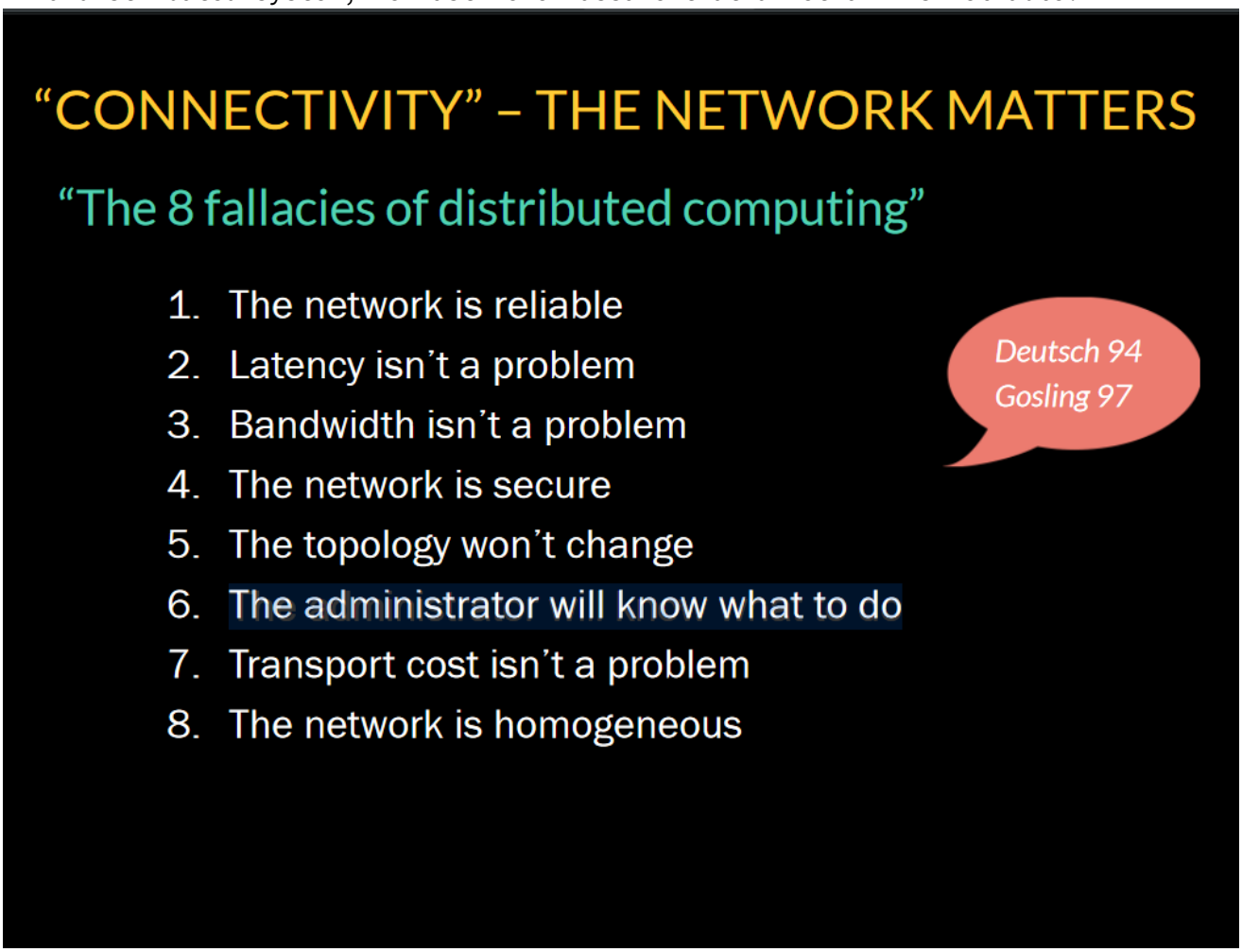


Image source

- The network is not reliable. The problem is that we're hiding the fact through service calls, and dependency injections. Given the system is chaotically interconnected, we end up with retries, slowly building up queue of requests that will eventually gush and lay down the whole system. Another common source of the misconception is that while using a cloud provider, we may presume we're safe, but while it's reliable on the whole, the cloud consists of many unreliable pieces. A server can be rebutted, and a security patch can be applied at any time, so our application has to be developed in a way to tolerate it.

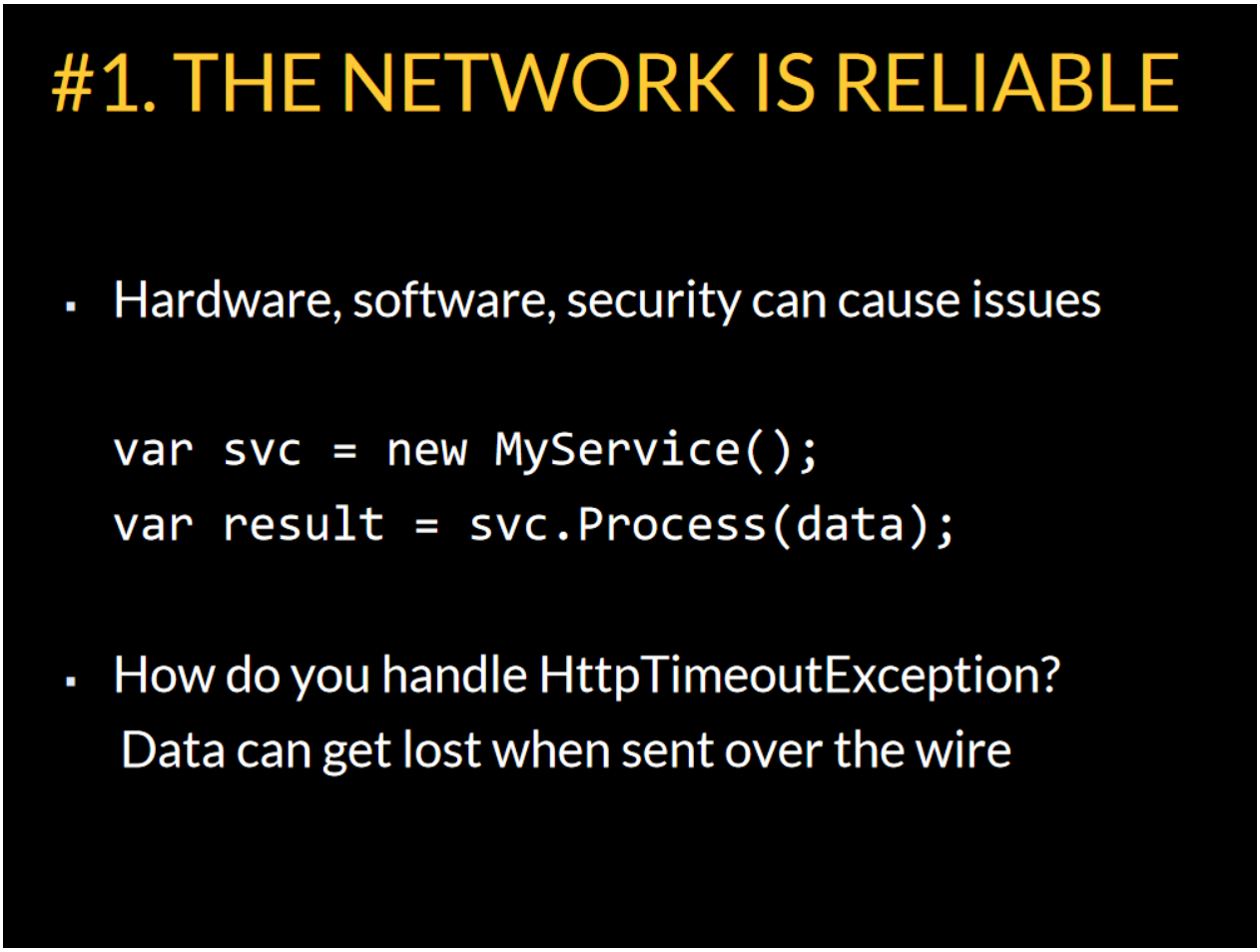


Image source

- Latency is an issue. We tend to think about things like network calls to be instantaneous, most of the time. The problem appears when we scale the latency numbers; "First Law of Distributed Object Design: Don't distribute your objects" because it isn't the same to call an object in memory and on the network. Entity Framework core, as other ORMs, would lazy-load additional data from DB without you knowing, adding a lot of latency, now, hopefully, you need to make it explicit

EVENT	LATENCY	SCALED
1 CPU cycle	0.3 ns	1 s
Level 1 cache access	0.9 ns	3 s
Level 2 cache access	2.8 ns	9 s
Level 3 cache access	12.9 ns	43 s
Main memory access (DRAM, from CPU)	120 ns	6 min
Solid-State disk I/O (flash memory	50-150 µs	2-6 days
Rotational disk I/O	1-10 ms	1-12 months
Internet: San Francisco to New York	40 ms	4 years
Internet: SanFrancisco to United Kingdom	81 ms	8 years
Internet: SanFrancisco to Australia	183 ms	19 years
TCP Packet Retransmit	1-3 s	105-317 years
OS virtualization system reboot	4 s	423 years
SCSI command time-out	30 s	3 millennia
Hardware virtualization system reboot	40 s	4 millennia
Physical system reboot	5 m	32 millennia

Image source

- Bandwidth is limited. No matter how many parts we decide to split the load among. The sizes of disks and the speed of the CPU grow much faster than the bandwidth and internet speed. With increasing latency, we want to prefetch, with the limited bandwidth, however, we want to fetch only necessary data, which makes us create smaller models tailor-made for tasks. Even if you consider this to be useless bites-counting routing, there is much more to the narrowly focused models than just performance.

Therefore, we need to understand how to build a reliable system over an unreliable network
By turning a blind eye, we introduce ourselves to the vicious circle of coupling.

> That's the reason why the idea of WCF (Windows Communication Foundation) is hilarious, it looks like a magic box, especially for freshman who just learns the concepts, that makes appearance like you're calling an object, but actually, you've just made an RPC.

The vicious circle of coupling

Starting from smaller concepts we have desperately been trying to orient our code into components each time attempting to simplify change and extensibility but failing again and again when coupling slips through:

GOTO -> procedural code -> OOP (encapsulation would introduce too much coupling in a big system leading to stiffness, inability to change) -> Components (encapsulation in components, not between) -> SOA (Service Bus is too smart) -> Microservices

As you can see, the coupling is the problem that crosses all the concepts, it isn't related to just the microservices [2]

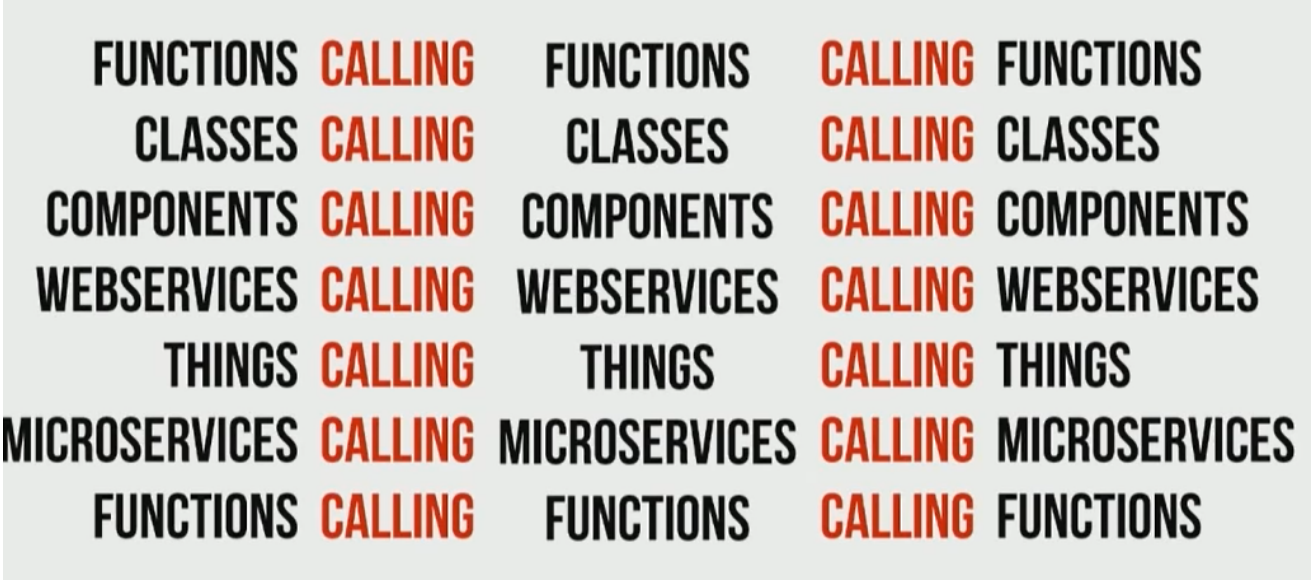


Image source

Without constraints, the coupling can be introduced to any system, will it be the OO one or Microservices. Distributed systems, however, are less prone to the problems just because of the physical separation that makes it harder to cross the boundary and break the constraint. [8]

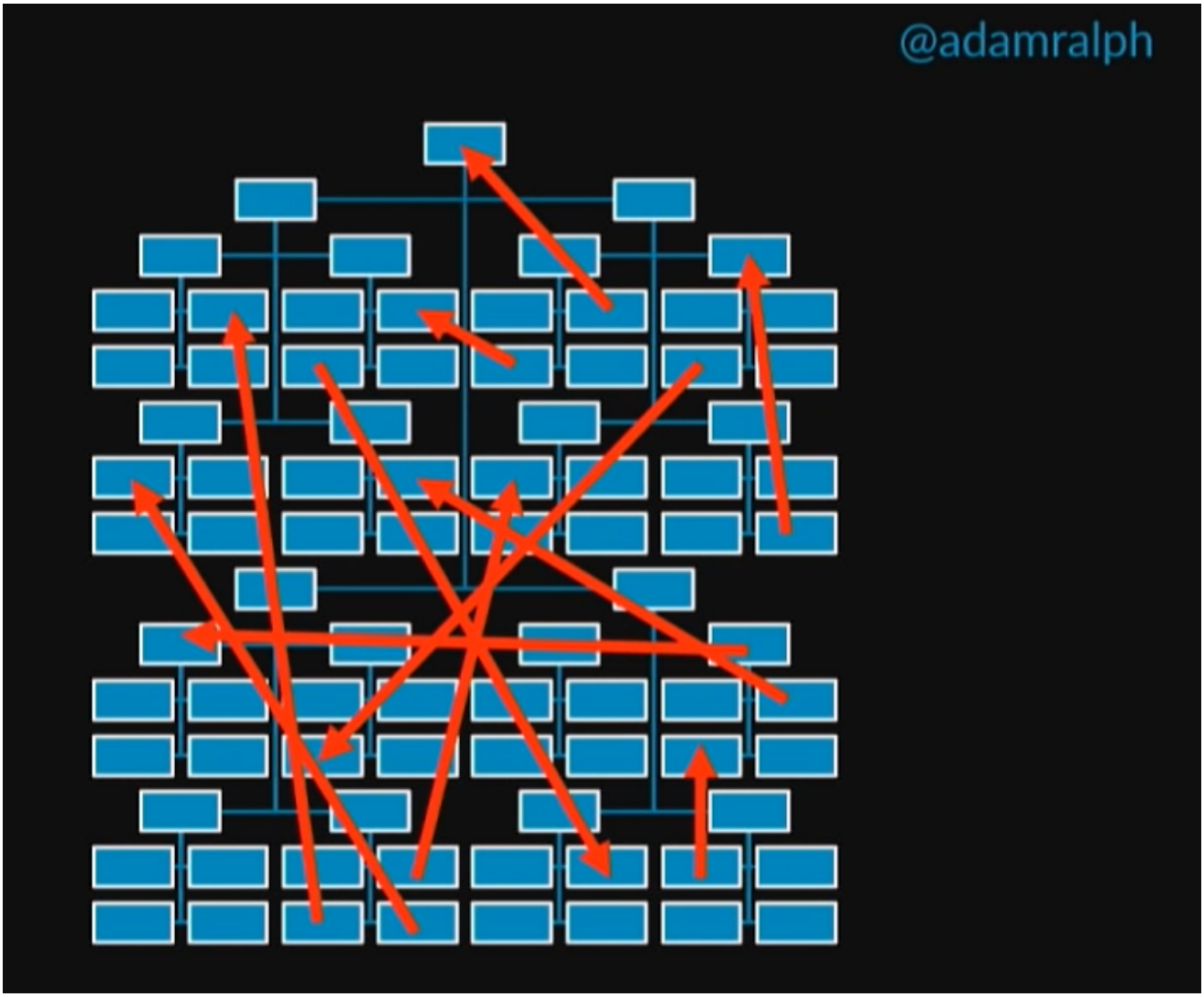


Image source

Reducing dependency

So, we discussed the problem and the cause, it's the time then to carry on with the principles we should build our systems upon to prevent most of the issues.

Establishing boundaries

The most important thing is to take control over sprawling features in a big system by carefully choosing boundaries.

In microservices, the boundary is the surface that introduces coupling, the microservice 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). 7

Image source

Identifying boundaries isn't a one-shot 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. 4

So, how to decompose? As we have already understood, the size isn't a good indicator.

“There are many useful and revealing heuristics for defining the boundaries of service. Size is one of the least useful.” - Nick Tune 11

We must break our system apart using cohesion as the guideline, not granularity, and DDD may help us with that. You see, in DDD there is a description of a very important concept called *Bounded context*, which is a separate unit in an application that is defined by domain meaning. 12

As Eric Evans said, if he had been rewriting the book, he would have moved the Bounded Context chapter to the beginning, because Building blocks like entities and repositories are overemphasized, and it prevents people from getting to the very end where more important information is presented. 3

This concept gives us the right granularity for our boundaries in a distributed system. Bounded context may be a single microservice or a group of such because it's not the physical but logical unit that is, however, bigger than classes', modules' and microservices' therefore easier and more useful to maintain. 2 4 9 13

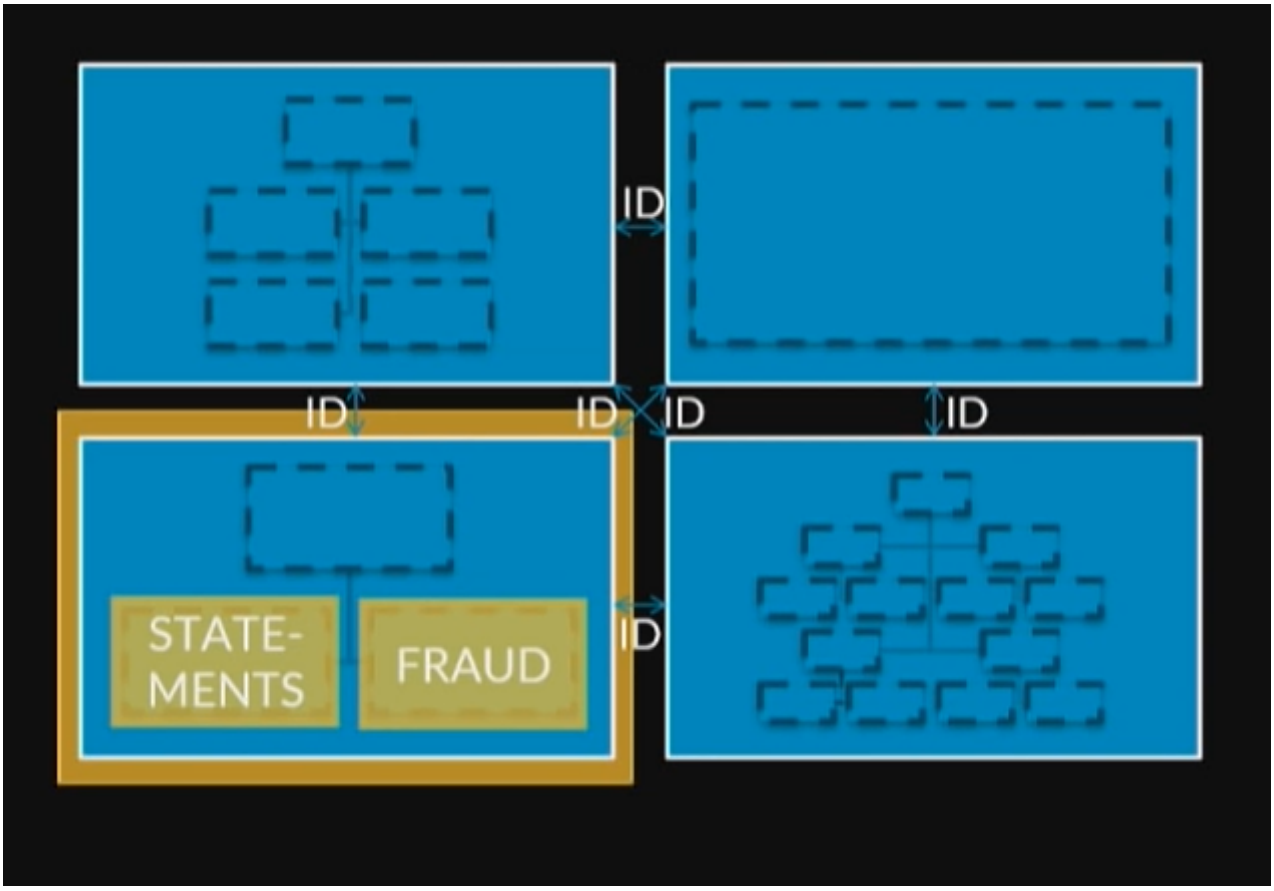


Image source

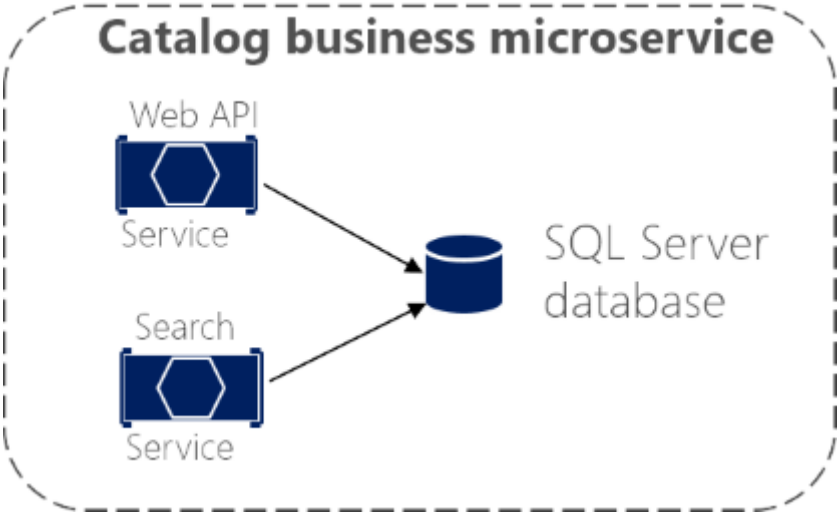


Image source

Bounded context 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 very 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 bounded context we allow coupling in, we know that we will need to make changes, we know that only one team is responsible for that, but changes mustn't propagate outside. 9 14

Applying to distributed systems, 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 has to be responsible for one or a combination of such units, removing dependency on other teams and increasing

agility this way. 2 9

Image source

Identifying the boundary

Bounded contexts should be split depending on the functional relation, and behavior and span across all the layers from data to view (utilizing the micro-frontend technique). That is, you do not split the context depending on technical requirements like the platform (mobile, desktop) or the type of process (long-running or not). It's similar to the way a project is structured, we group by feature, not by the type of file, like controllers, DTOs, or services... This is essential to ensure the absence of collisions between different teams during the development process. 2 9 12

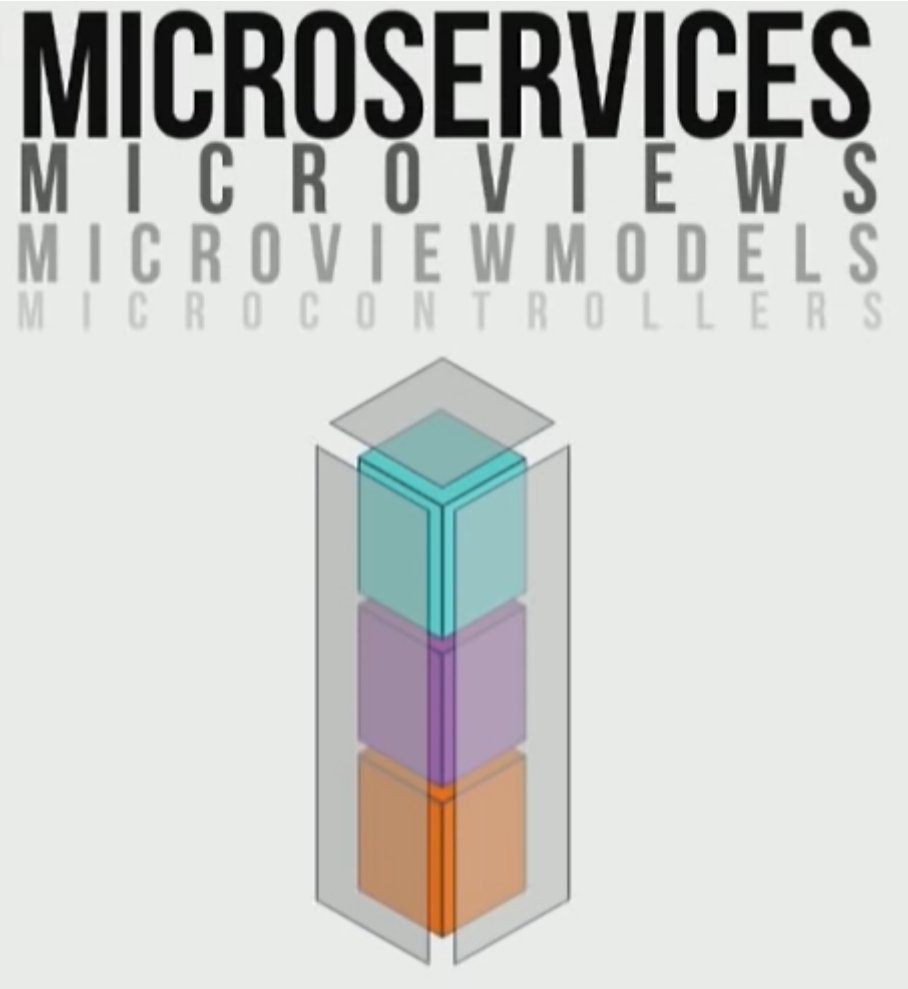


Image source

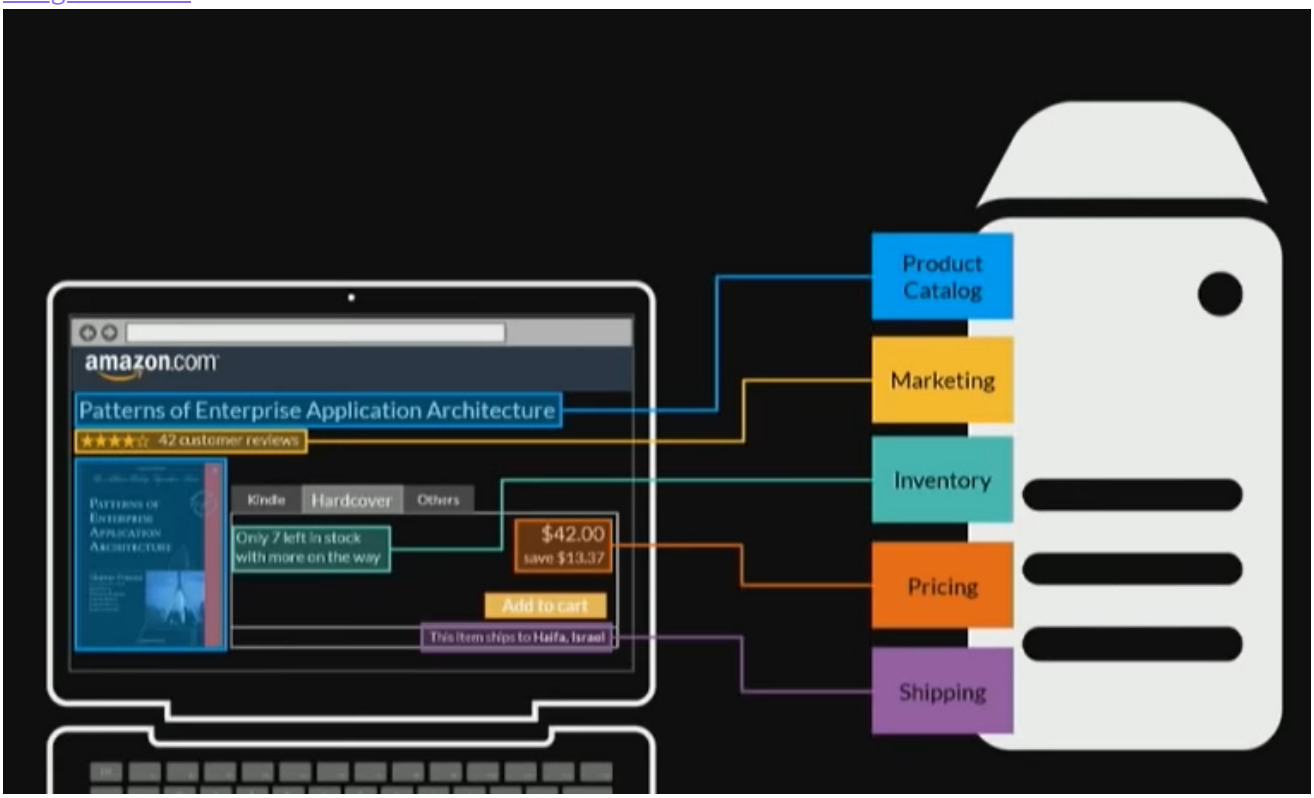


Image source

As a hint, domain language can be used to identify separate contexts. As an example, we may have the same user identity across the whole application, but in different 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. Also, different rules are present, so the separation of those parts is important. [10] [12] [15]

Image source

The three-way relationship

It is as important to form our teams the right way. There is *Conway's law* that states organizations design systems that mirror their own communication structure. Without getting our organization's structure in order, we won't be able to get rid of the chaos in the code. [11] [16] [17]

Therefore, we may see a three-way relationship between domain, architecture, and teams. When organization structure changes under the new discoveries, we need to ensure that we also update our boundaries in the code and team structures accordingly to prevent cross-contexts teams. [11] [12]

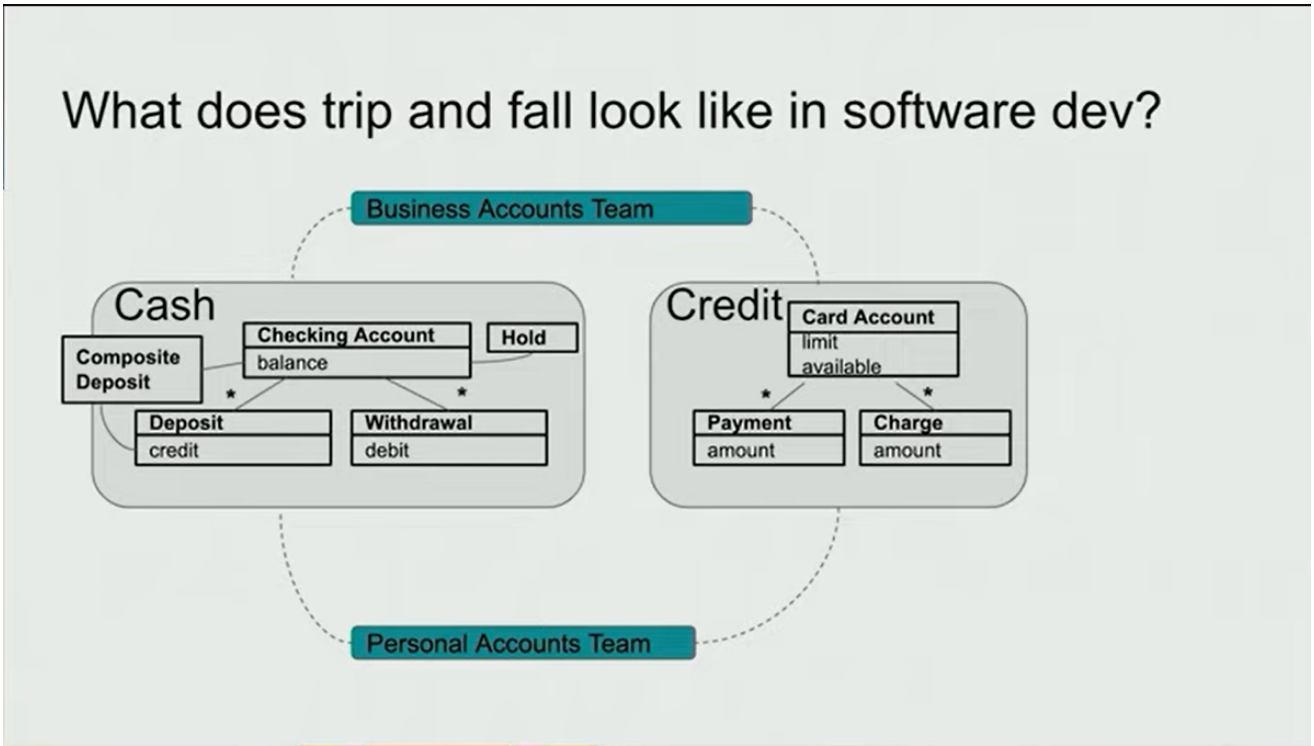


Image source

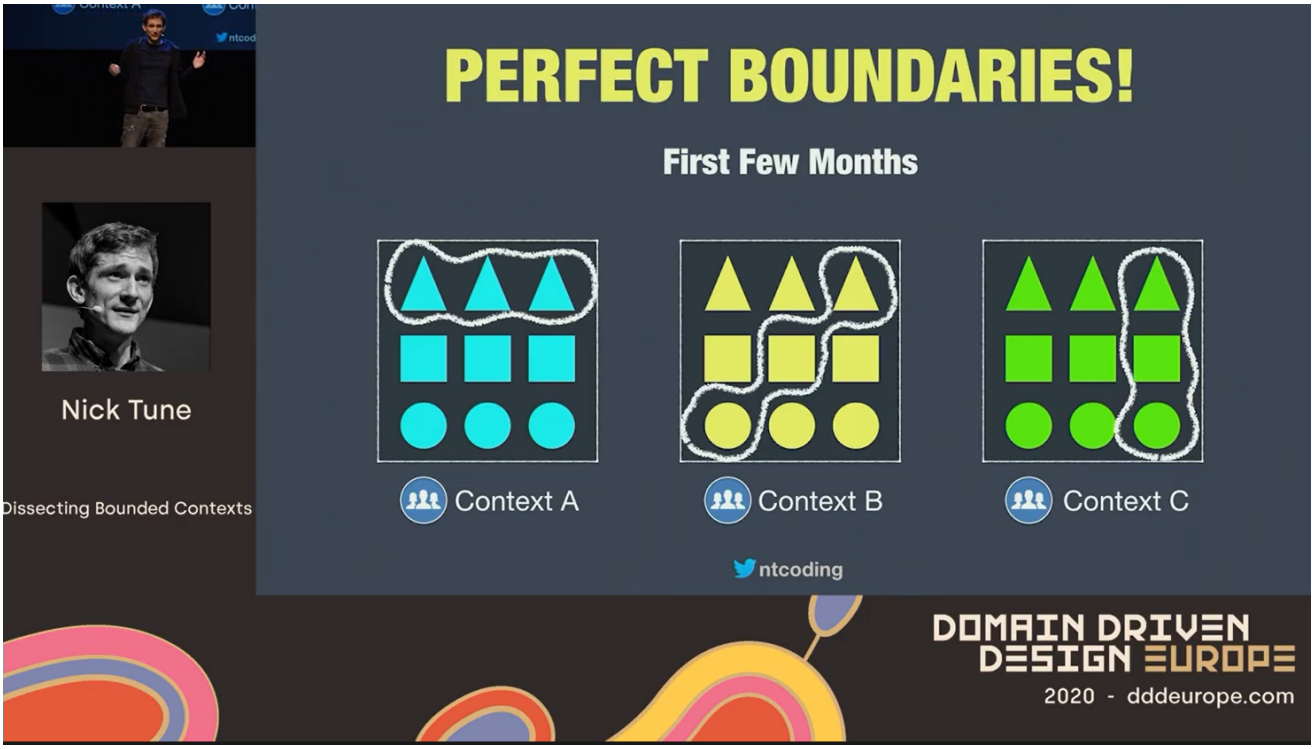


Image source

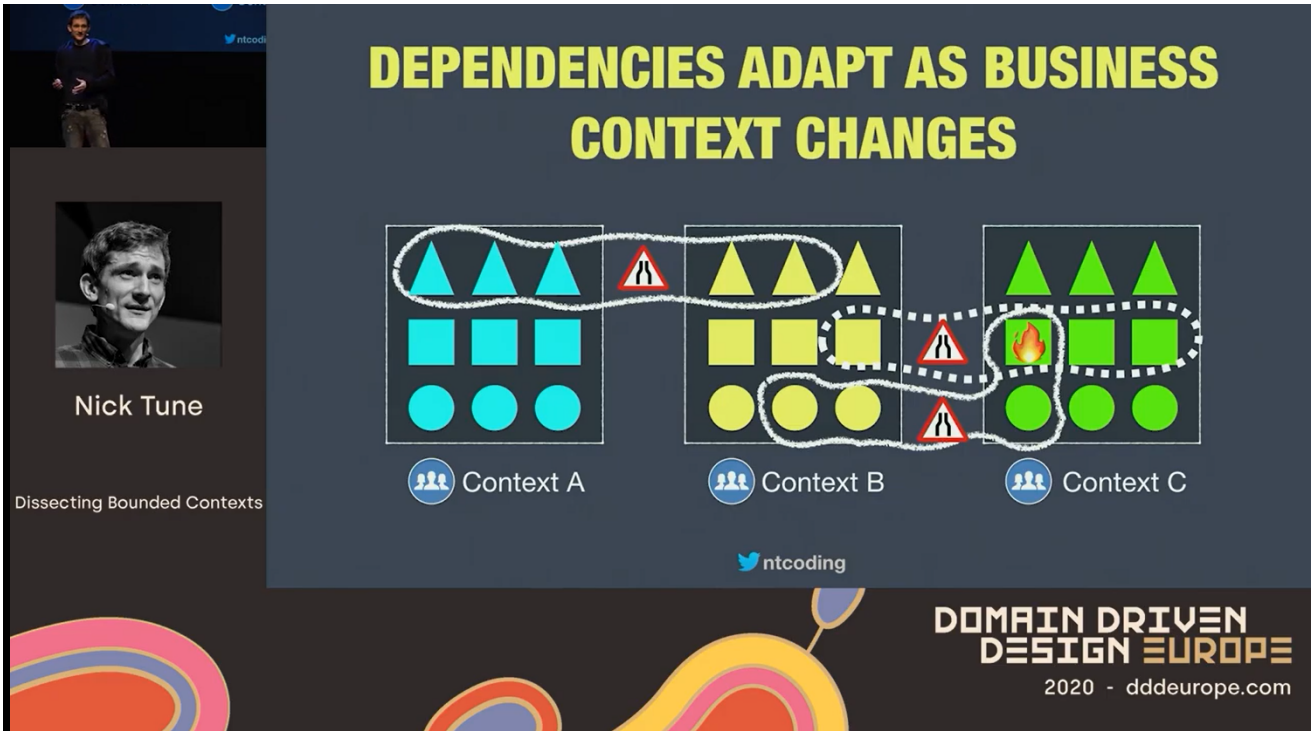


Image source

How domain influences bounded contexts

There are many heuristics related to the way of defining a good boundary of a microservice, 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: 10 11 18

- 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 the part of the domain that you do not want to quickly decompose into microservices.
- Supporting - support core ones and contains the functionality specific to the domain that is not crucial but the app can't work without.

What the separation gives us, would you ask? It provides a clear map of what to concentrate on, where to use complex approaches to modeling, and where to use DDD after all. Design approaches like DDD have their benefits but they add much complexity as well. There is no point in DDD for a CRUD. When we introduce complexity we only make it harder. 3

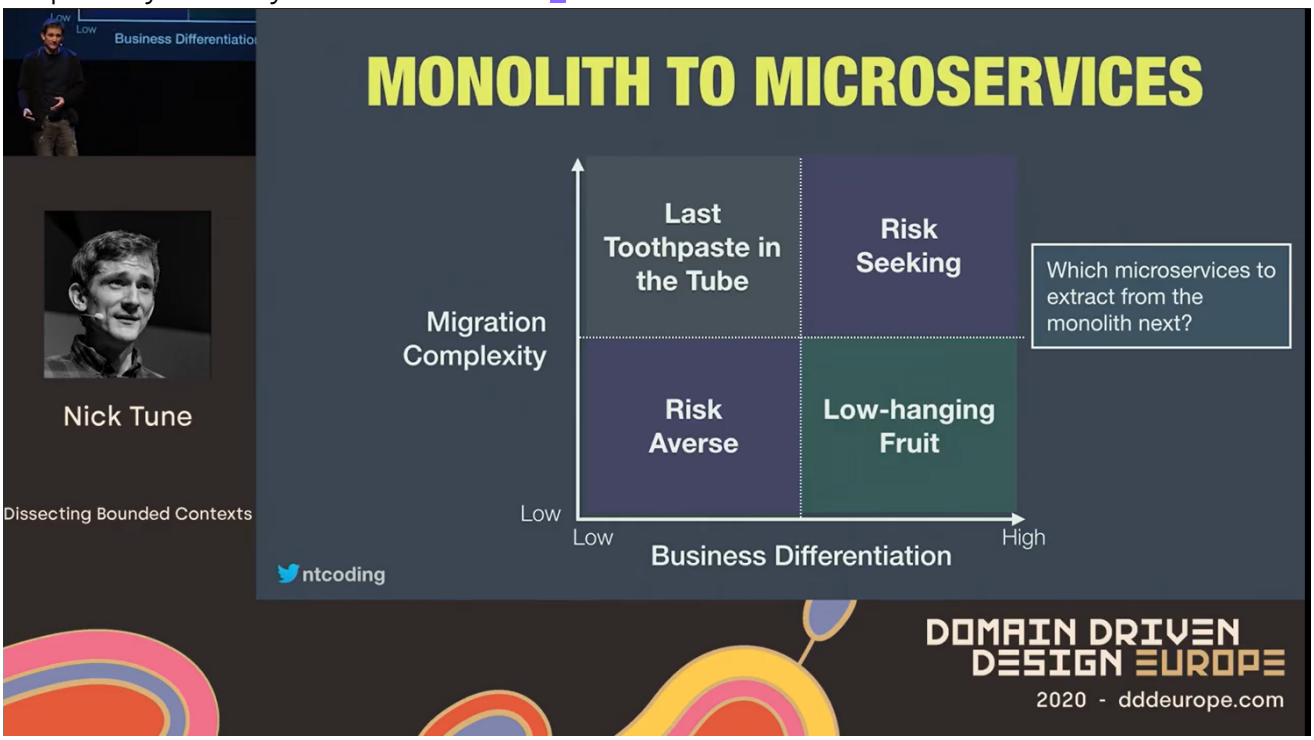


Image source

Identify subdomain

There are a few hints to identify the importance of an application part, all related to complexity: 18

- 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. 18

The complexity of application in relation to subdomain

Tunning complexity of part of an application to match the business value is vital: 11

- When the significance of the part of an application isn't that big, because competitors can easily catch up to us, we may simplify

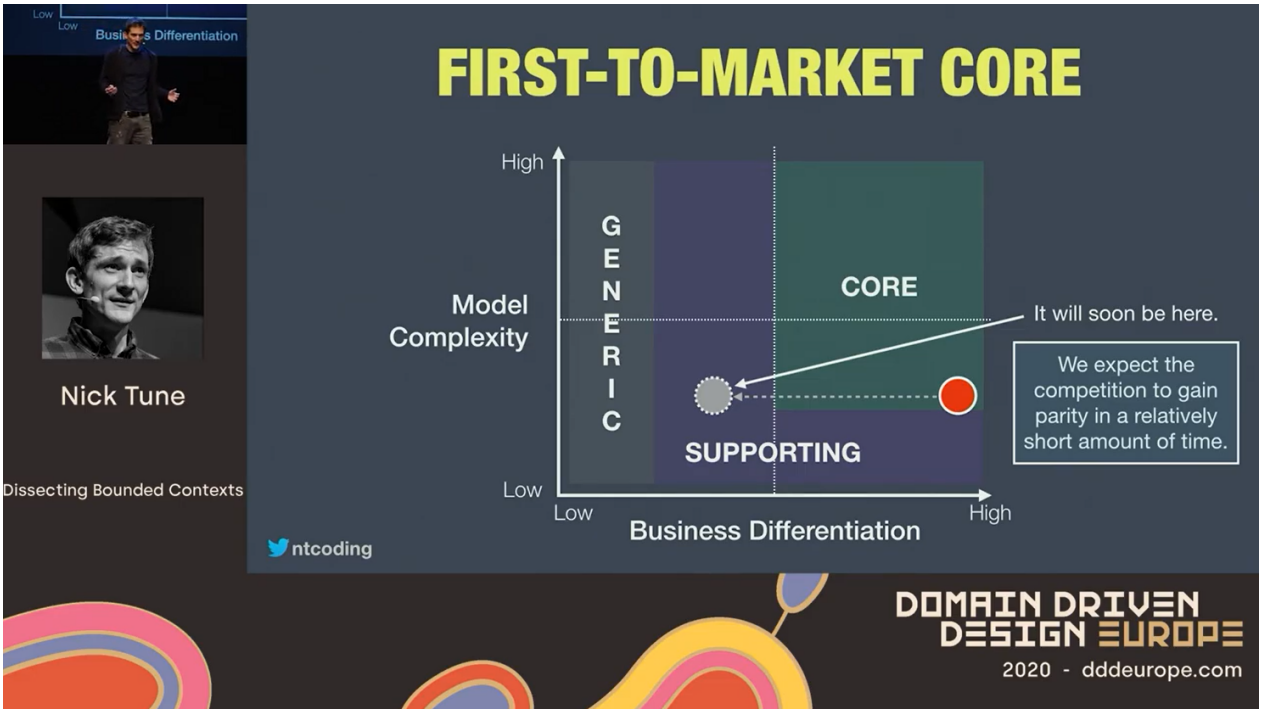


Image source

- When we under-investing time and effort into developing the part of an application that stands out and isn't that easy to catch up to

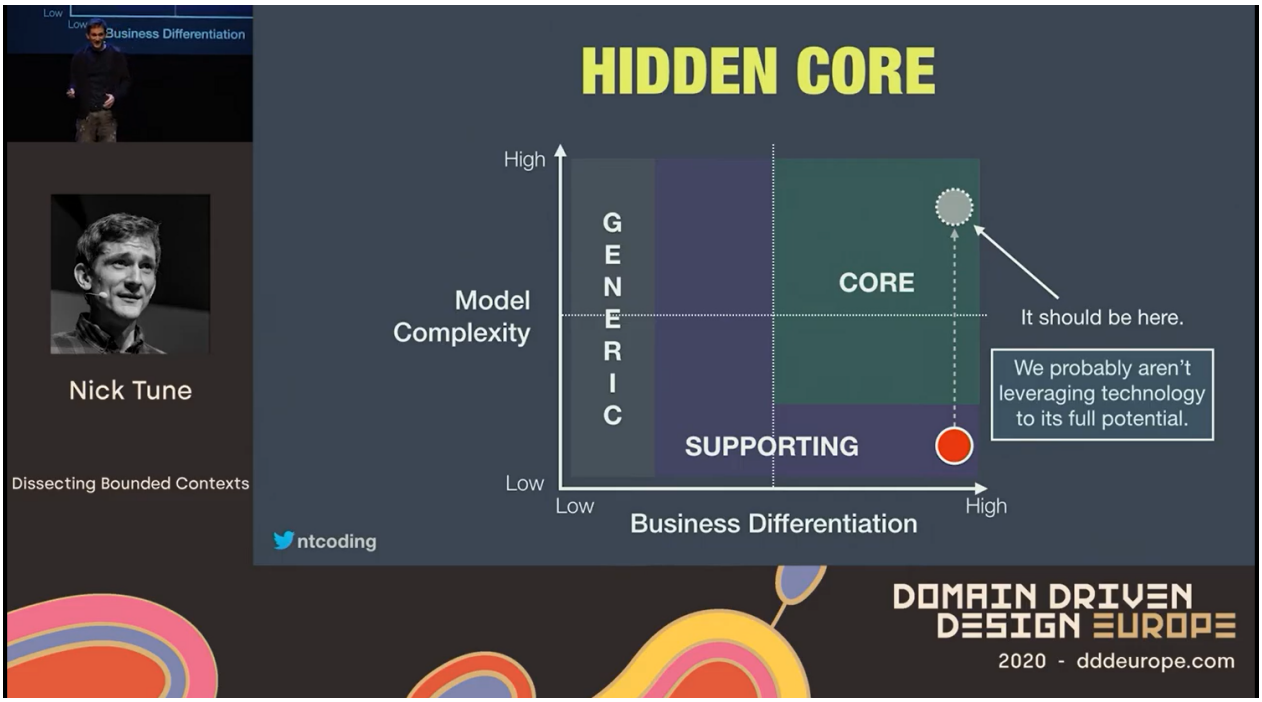


Image source

- When all the competitors catch up, the feature becomes just the expected one, leading to the shift of focus

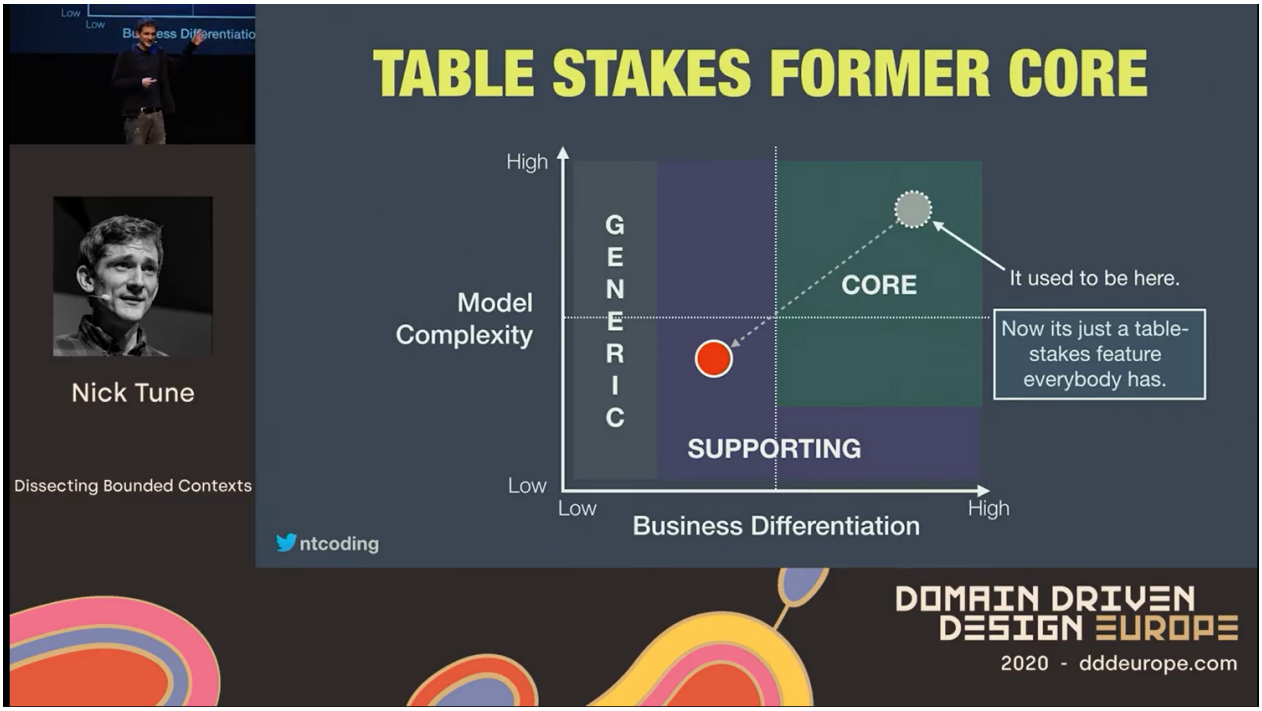


Image source

- Simplify all except the core. The complexity is probably accidental.

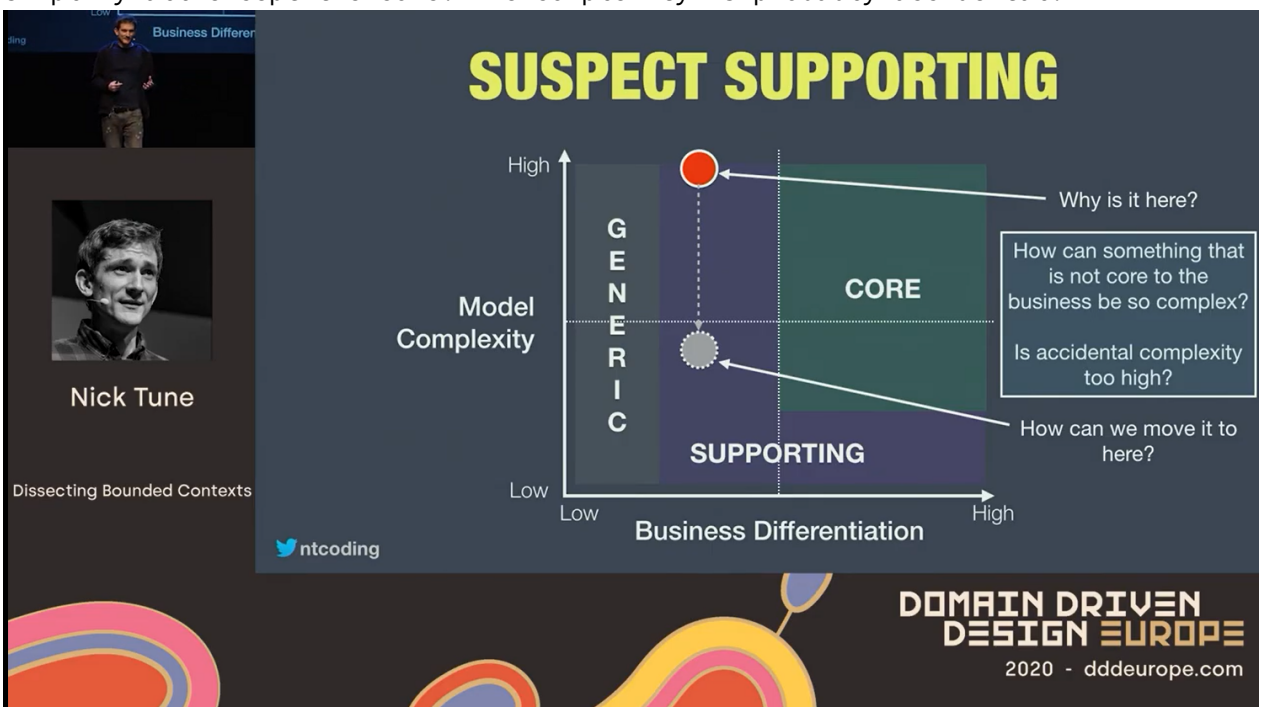


Image source

Domain always changes. Once generic part may become the core one. Like 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. 11

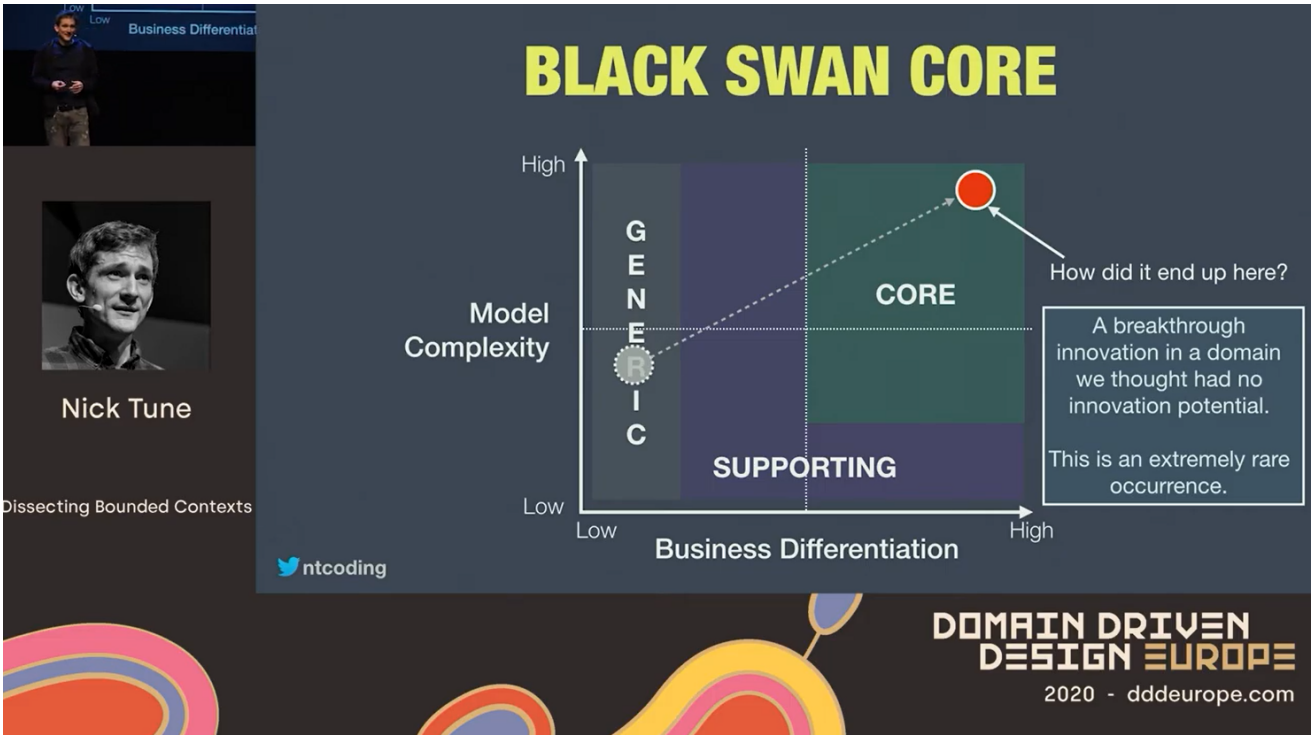


Image source

How data influences bounded contexts

Sometimes you can't make a feature work without some data, once independent units become quite chatty. The reason may be that you put it into the wrong bounded context, the wrong microservice, so redraw the boundaries. Other data-related heuristics are:

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

Interactions between bounded contexts

Interesting things happen not in bounded contexts but between them. If something goes wrong in one context, it must not cause other bounded contexts to stop working as well, this way we achieve resilience in our system. But it's an ideal case. Most often we deal with different kind of relations between the components. 19

Using a *context map* we can show logical relations. Arrows indicate the dependency and there are at least three kinds of relationships: 19

- 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. From the example below, we conform to the way the "Sales Force" works and embrace the same approach in our small system. This also means that we're accepting the need to change our application domain structure when the Sales Force's changes. Generally, 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, we will speak about it later.

- When we choose not to introduce such a dependency, we create an *anti-corruption layer*. 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 ok as long as we strive to protect our structural part from other parts because 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's an effective way to decouple legacy monolithic applications (aka *strangler* pattern [20] [21]) or integrate with third-party services.

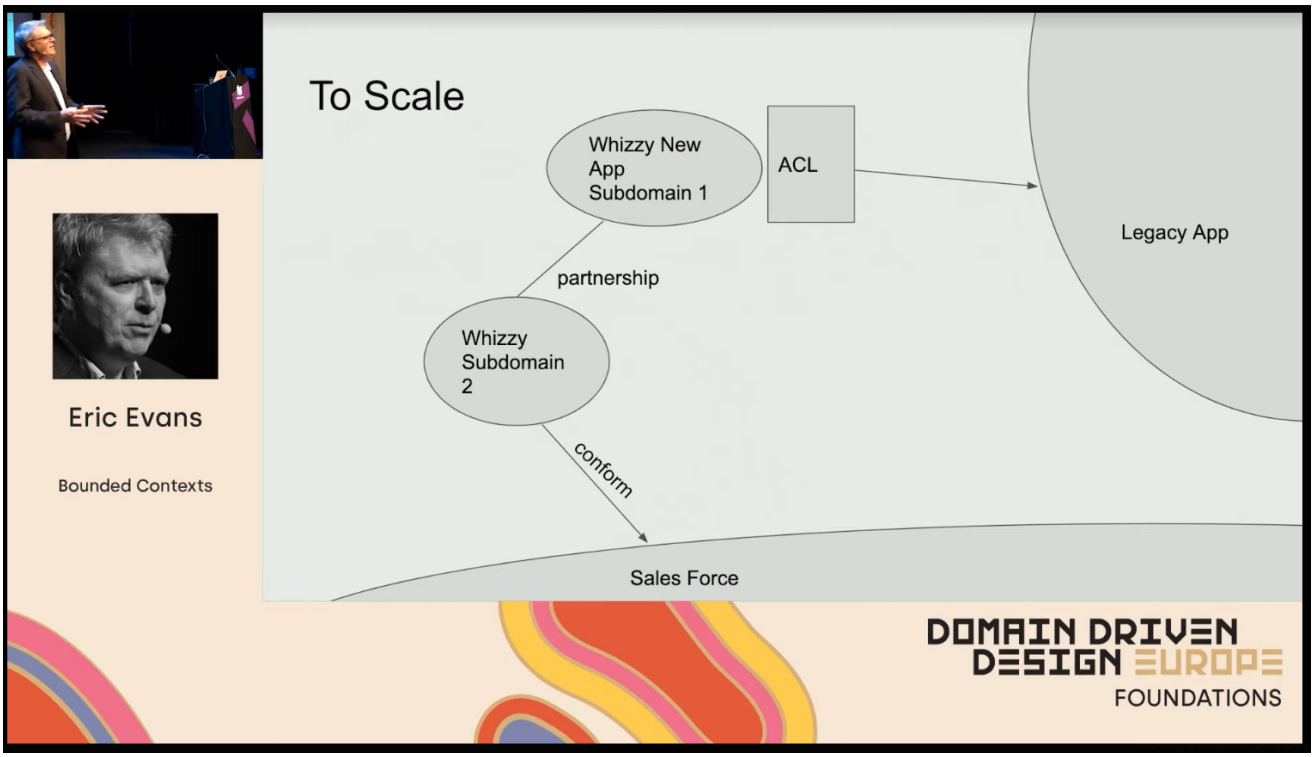


Image source

On the slides below the upper part is the physical communication between services and the bottom one is the context map which helps in identifying relationships. The arrows on the upper image indicate the flow of messages and the domain language that is used during communication. The context map shows us the logical relation.

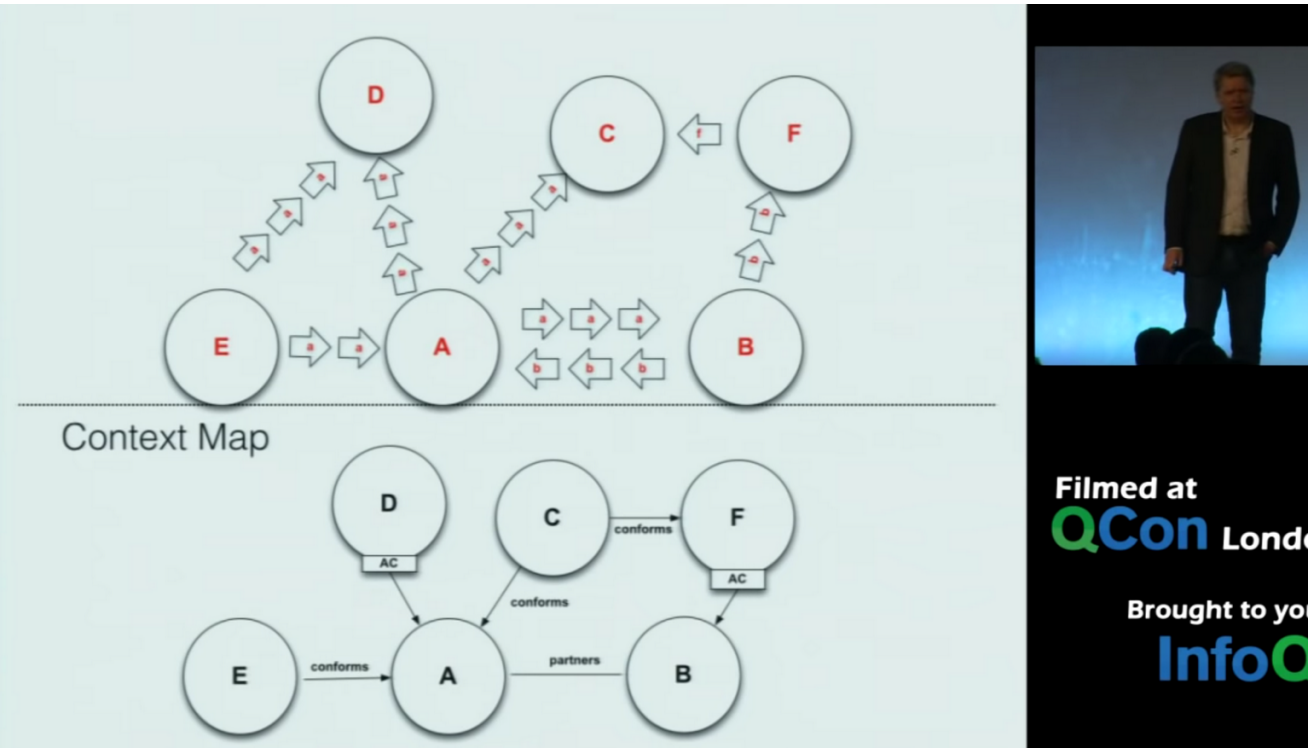


Image source

If we have an error in F and E contexts, it doesn't propagate far

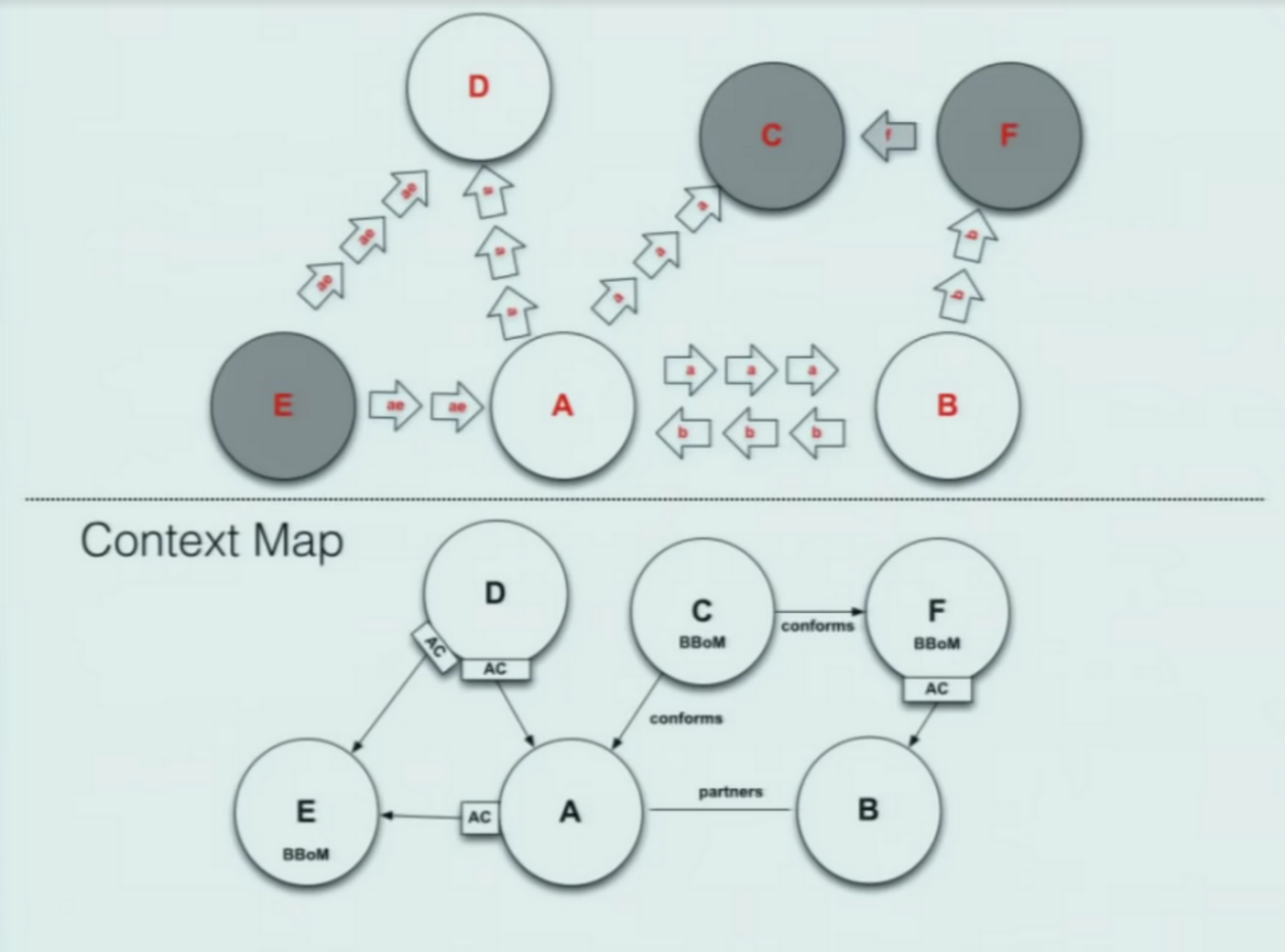


Image source

As we can see from previous examples, language A was used by 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 publish changes, not domain events

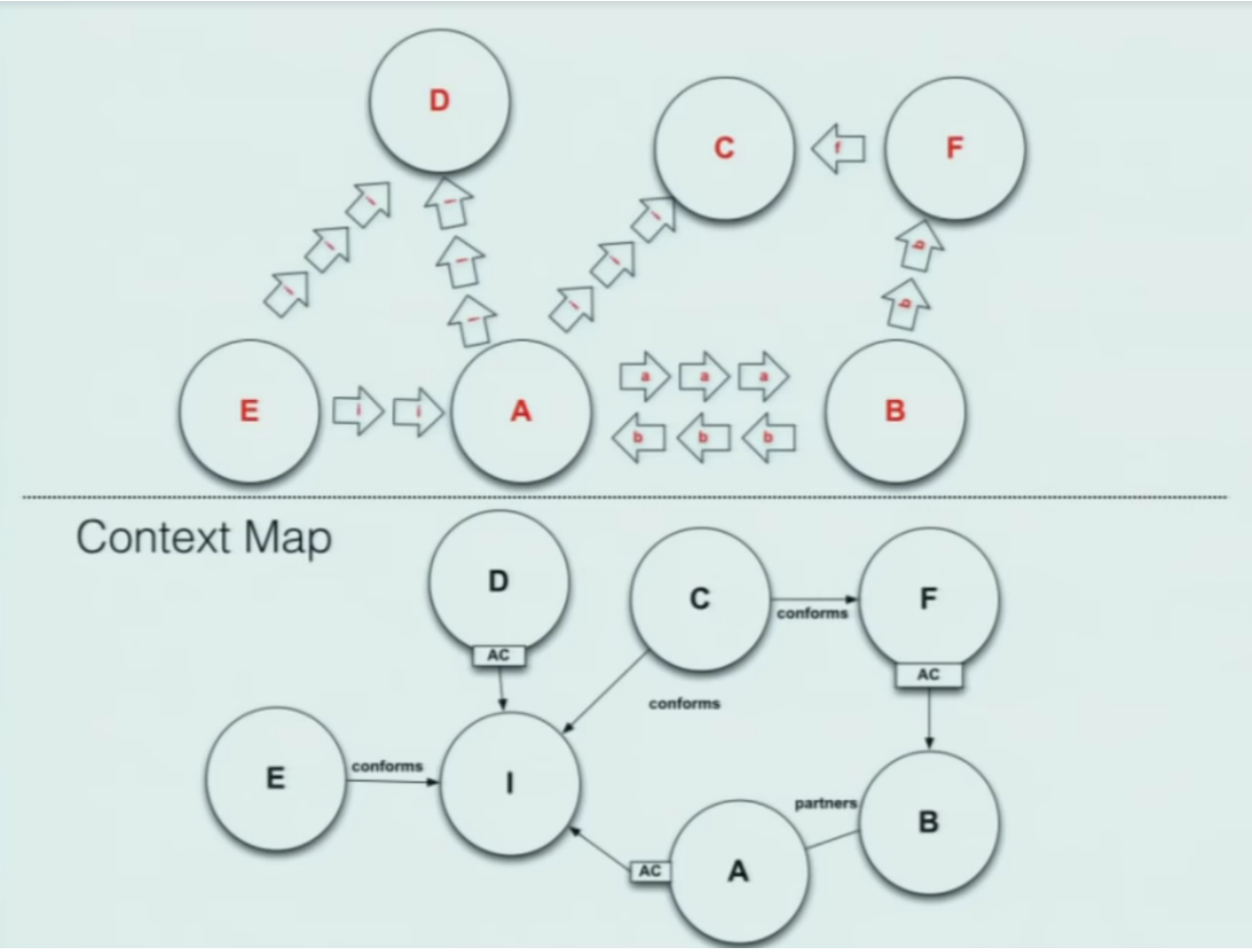


Image source

Logical boundaries within a monolith, are too permissive, so tend to break. The same goes for bounded contexts, that's why we need to be conscious about the boundary.

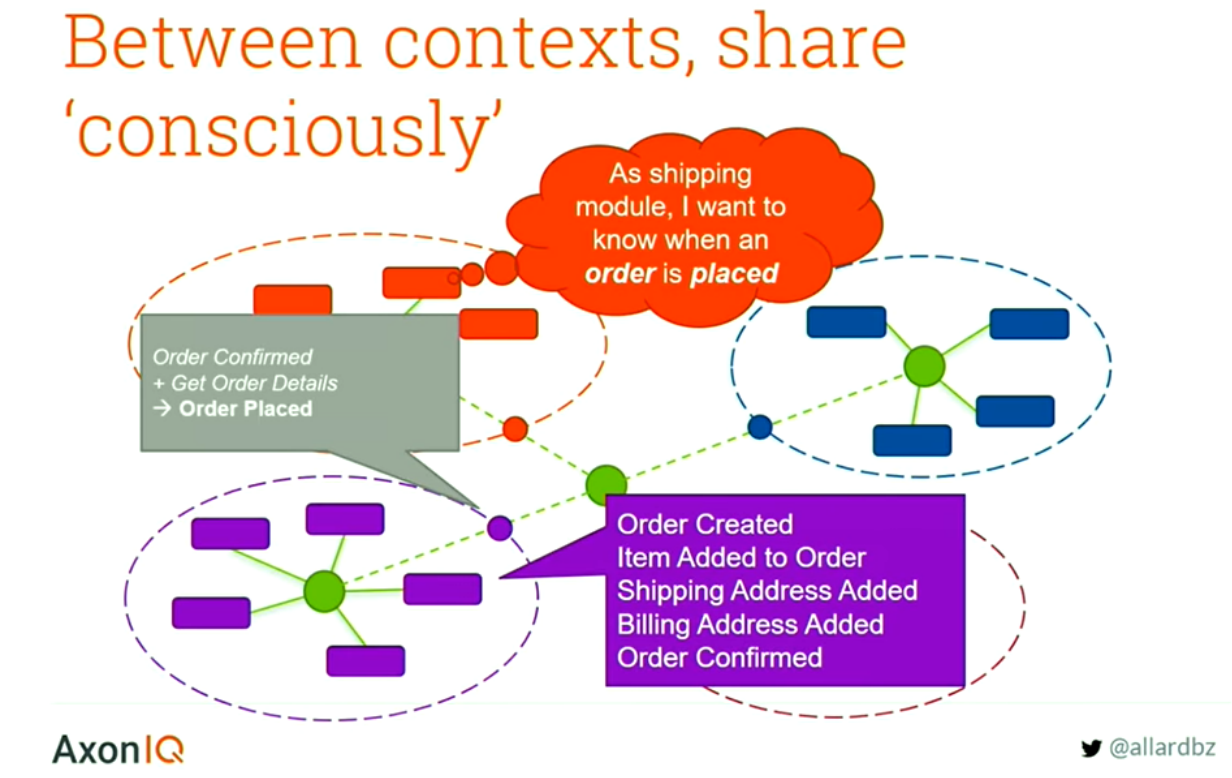


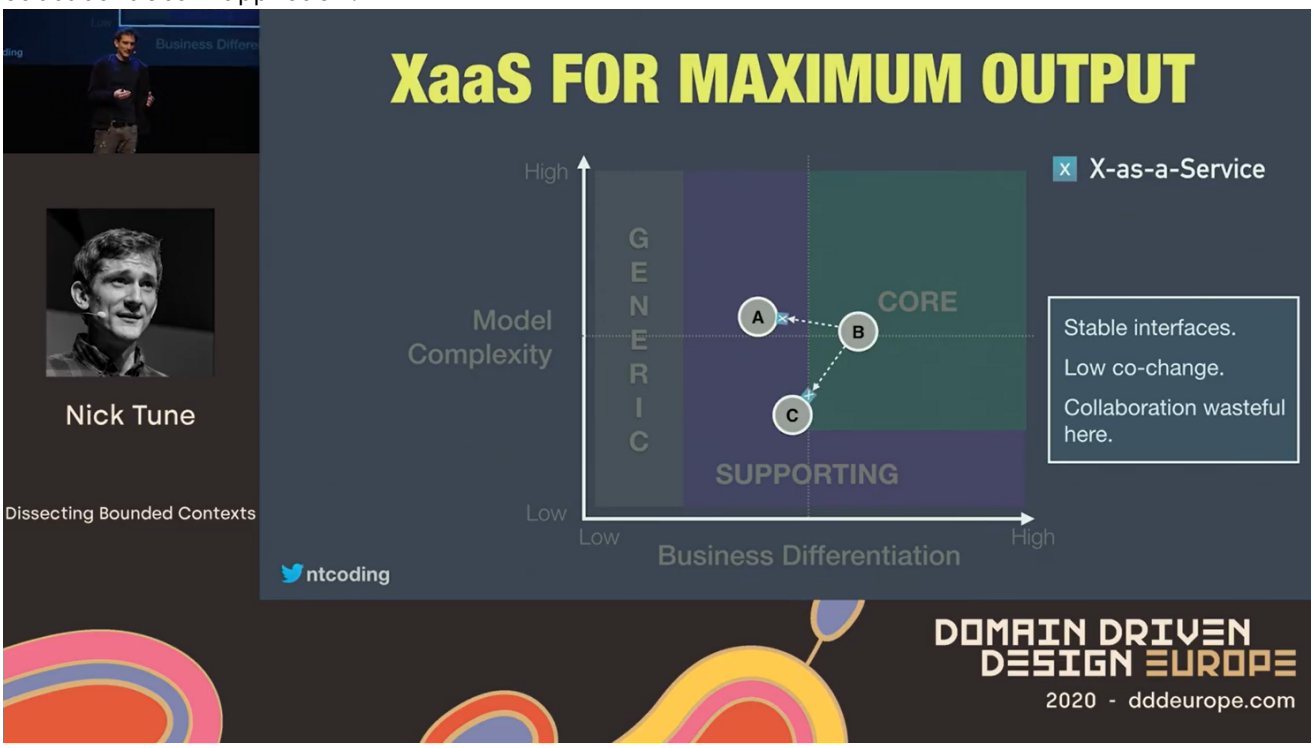
Image source

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 has to be developed quickly. *X-as-a-Service* removes the strong dependency and synchronization given by the close *collaboration* approach.



[Image source]

Otherwise, when we're exploring the core domain, the boundaries aren't clear, and multiple teams work together and *collaborate*. The core domain should be broken into services much later because in this stage much is unknown and it's required to iterate quickly to get the result. Moreover, one team may not have sufficient knowledge of the domain to finish the task therefore teams have to collaborate closely.

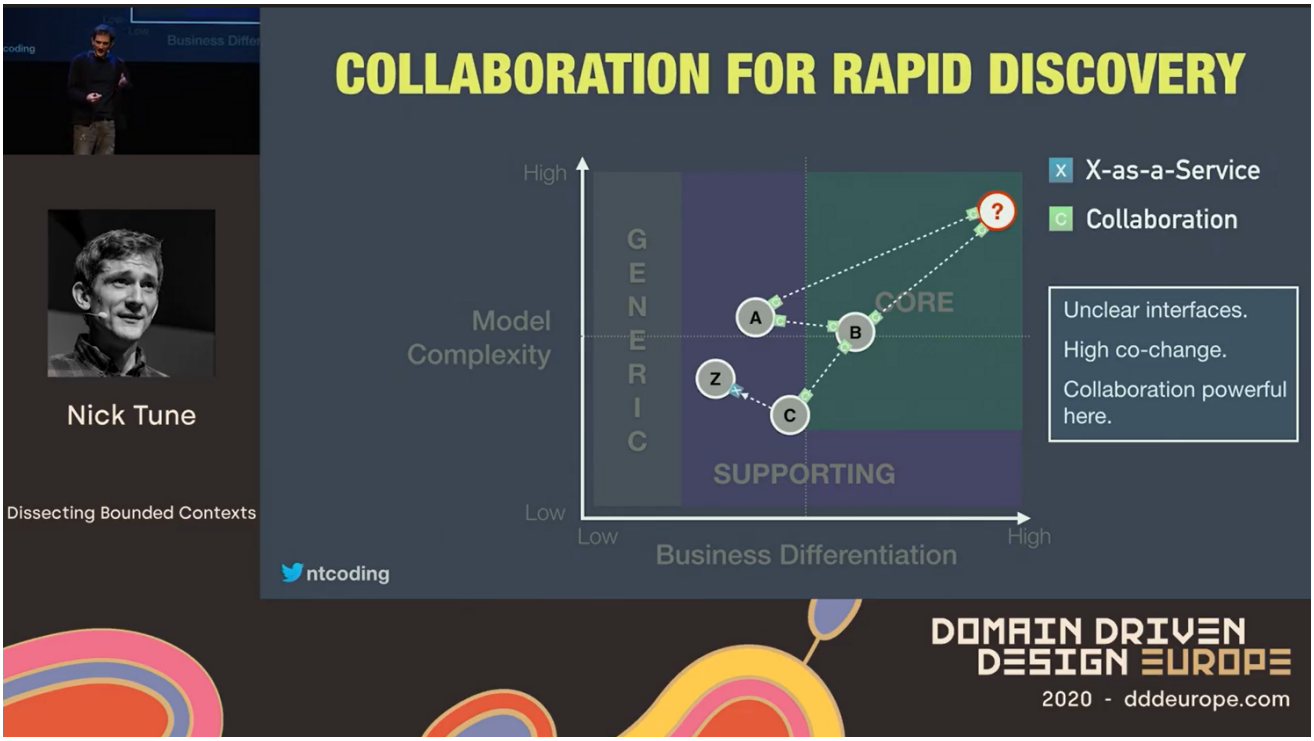


Image source

Depending on the stage of development of a particular part, collaboration approaches may transition from one another.

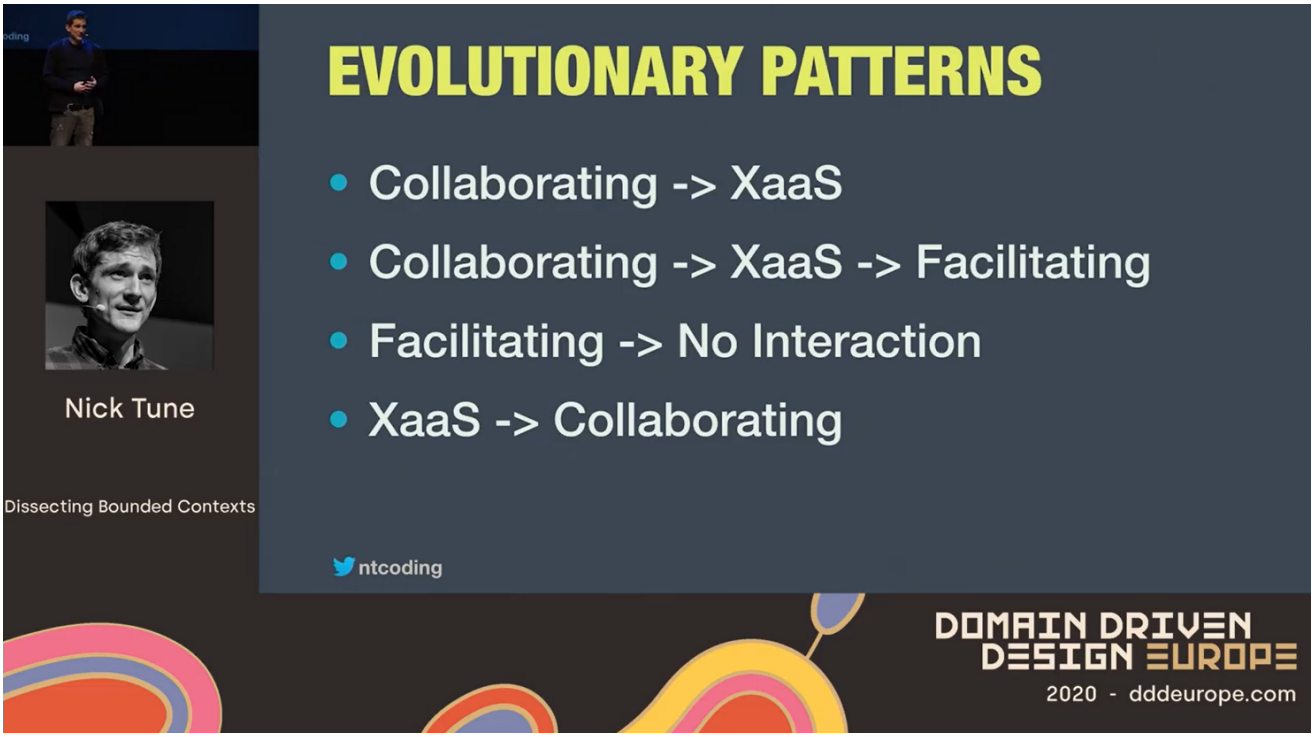


Image source

Teams are more flexible than application architecture, which can be employed to create temporary teams: 11

- for experimentation
- move people between teams for people to get a wider view

Boundary modeling techniques

There are techniques for modeling and identifying bounded contexts:

- Domain storytelling - helps to understand the bigger picture

- Event storming - helps in reviewing boundaries, extensive modeling

Practical examples of modeling contexts: [15] [22] [23] [24]

Sharing data

I'd argue that the data is 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 part will give you food for thought about the ways to consciously share data.

The purpose of sharing is driven by the business

The first and foremost goal is to understand what the data is used for. It may be utilized 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 theory, it's either one or another, we can't build a strongly consistent available system. As a rule, when strong consistency guarantees are required, the parts should go into the same bounded context, we will talk about the reason for that later. Most times it depends on the domain, as an example we need to ask experts whether it's ok 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. [25] [26]

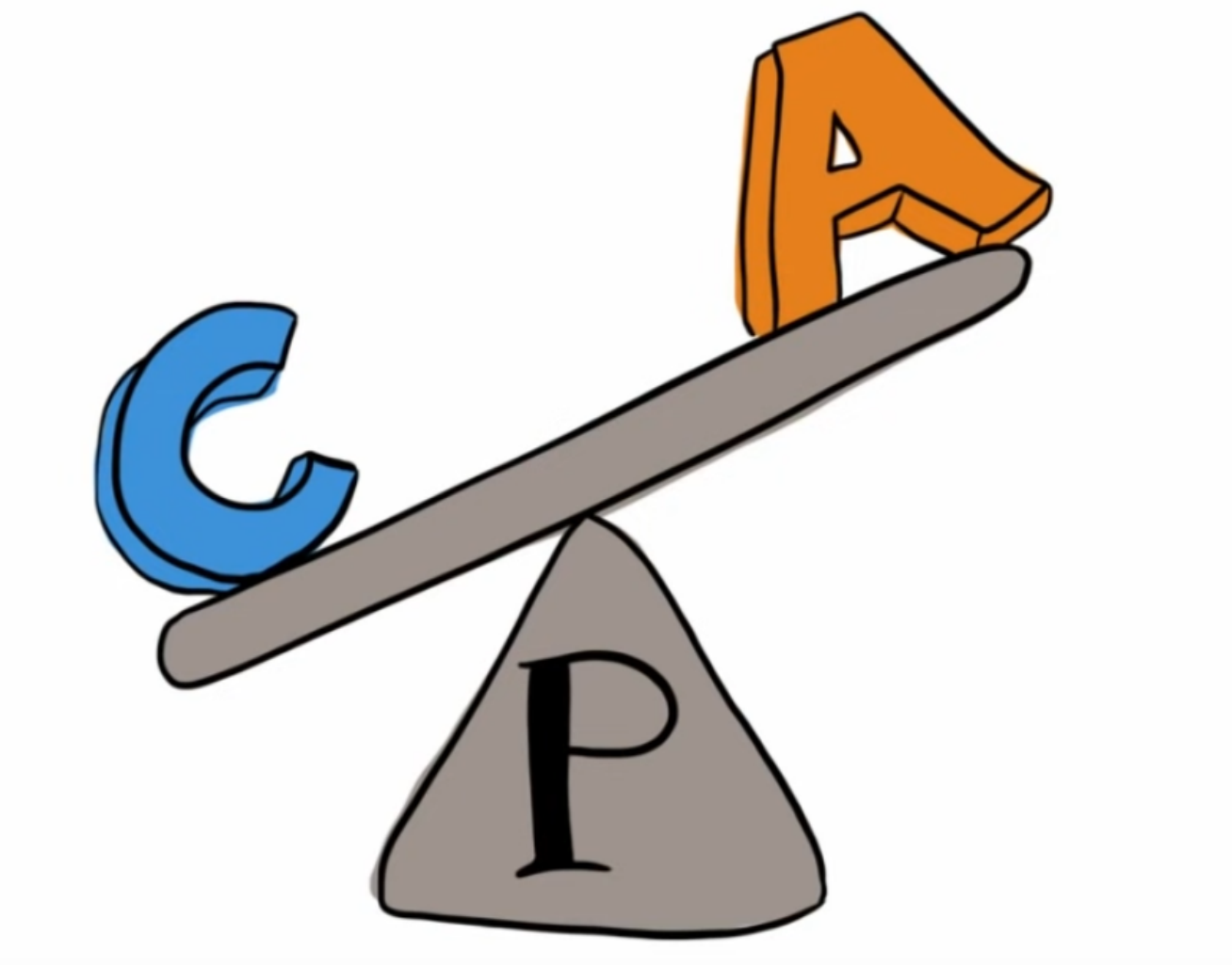


Image source

Data ownership

Another major characteristic related to data is ownership. As usual, there is no fast and hard rule and everything is greatly dependable on the domain. Once boundaries are established it's important to assign data to particular contexts 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 located in a separate context.

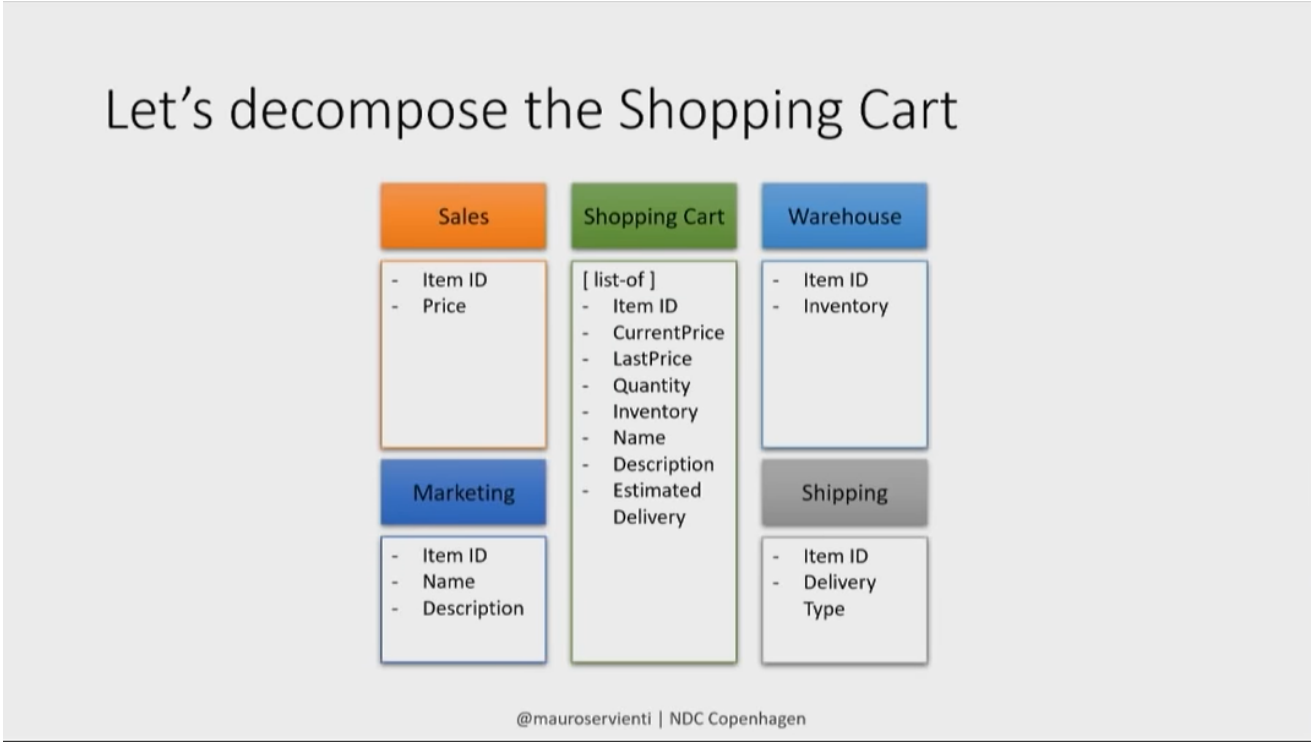


Image source

But the problem is that the shopping card concept just aggregates data from other contexts, essentially creating a view (cache), which, among other things, couples the shopping cart context with all the other ones introducing too much complexity.

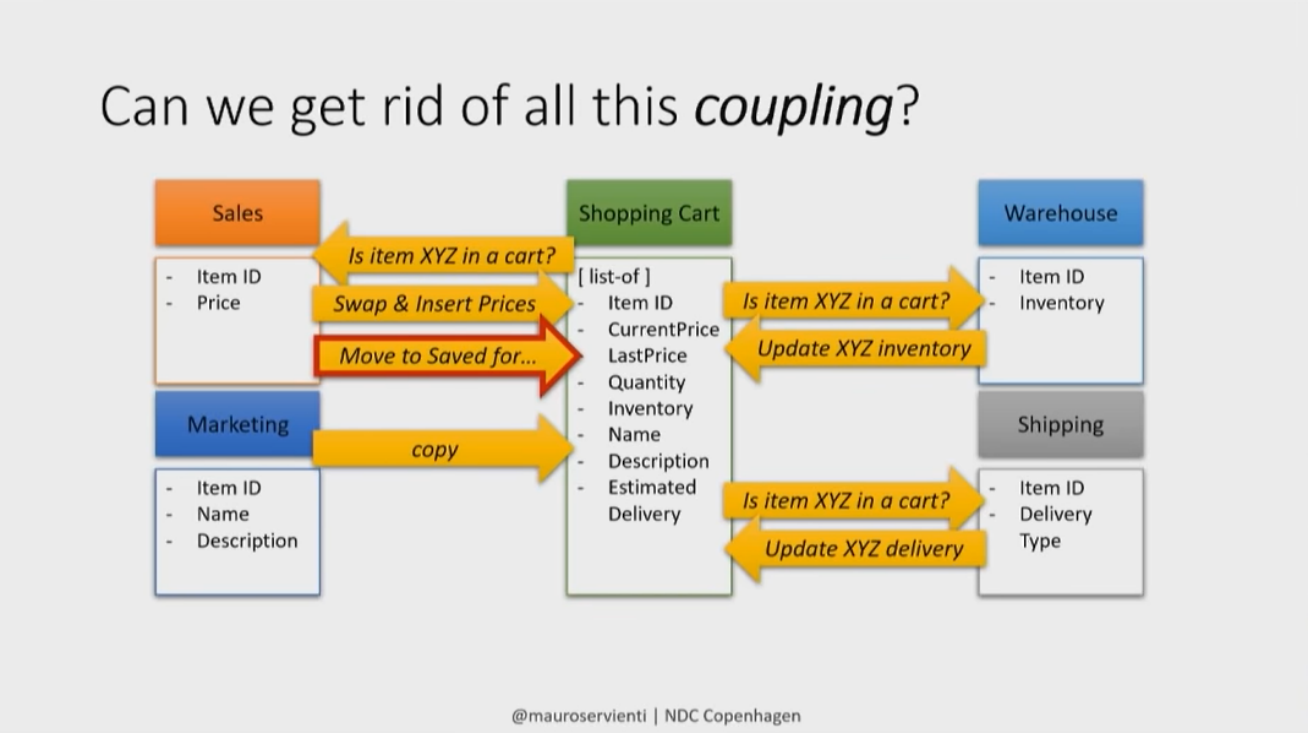


Image source

What we want is to have each context represent the part of a shopping card.

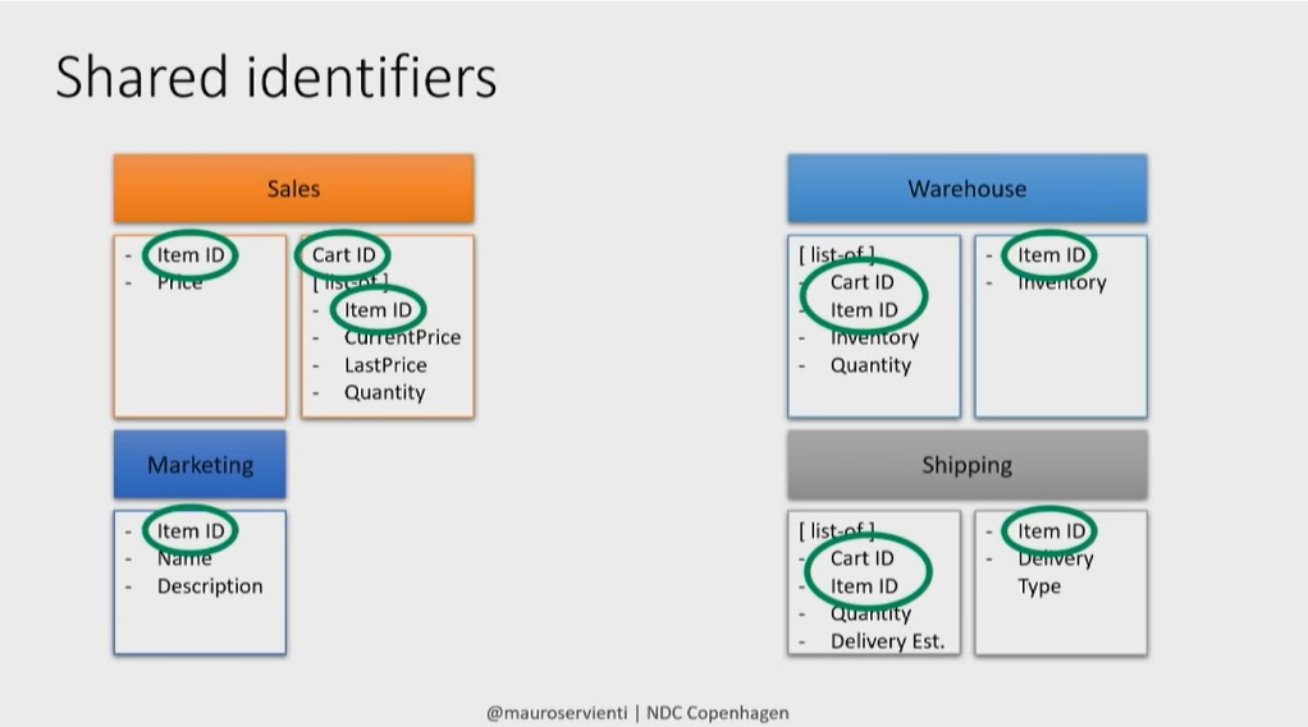


Image source

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 you want to share it or have only one service to own it), live in shared DB, in configuration, or be extracted in other microservice. 28

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

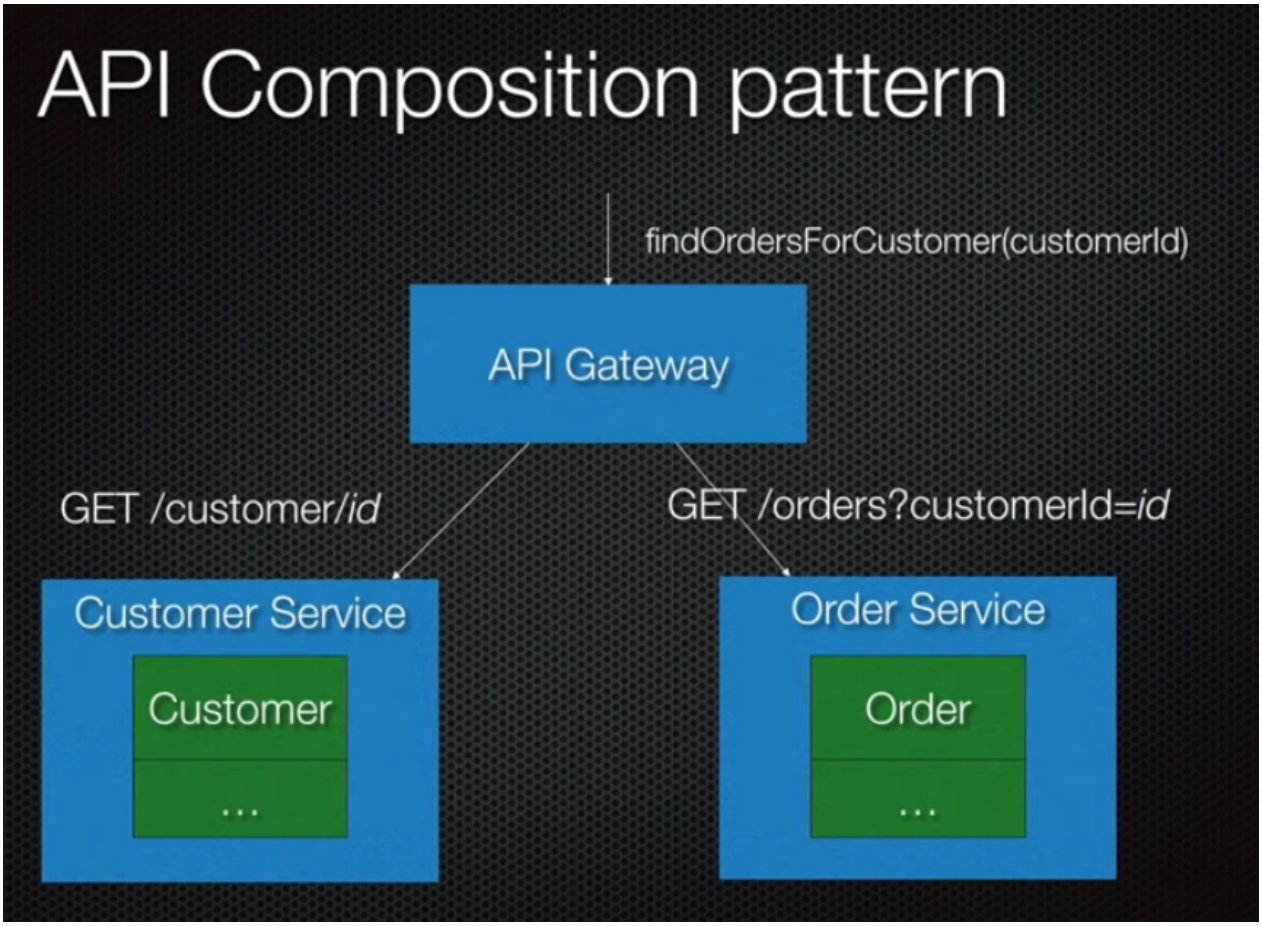
Reconsider talkative components

Check if it is being extensively shared with another service. If so, probably you'd rather redraw boundaries and merge the two services.

Data aggregation gateway

When we need to get data from different sources (microservices, bounded contexts). Depending on the goal you want to achieve (analytics, display on front end) and the amount of coupling you can tolerate, there are at least those approaches

- API composition (aggregation) pattern - combine the data using a gateway. Is not that suitable for situations that require joins over a big amount of data (for simple cases we can join in memory though). When *N+1* problems occur, other approaches like CQRS are more suitable. 29



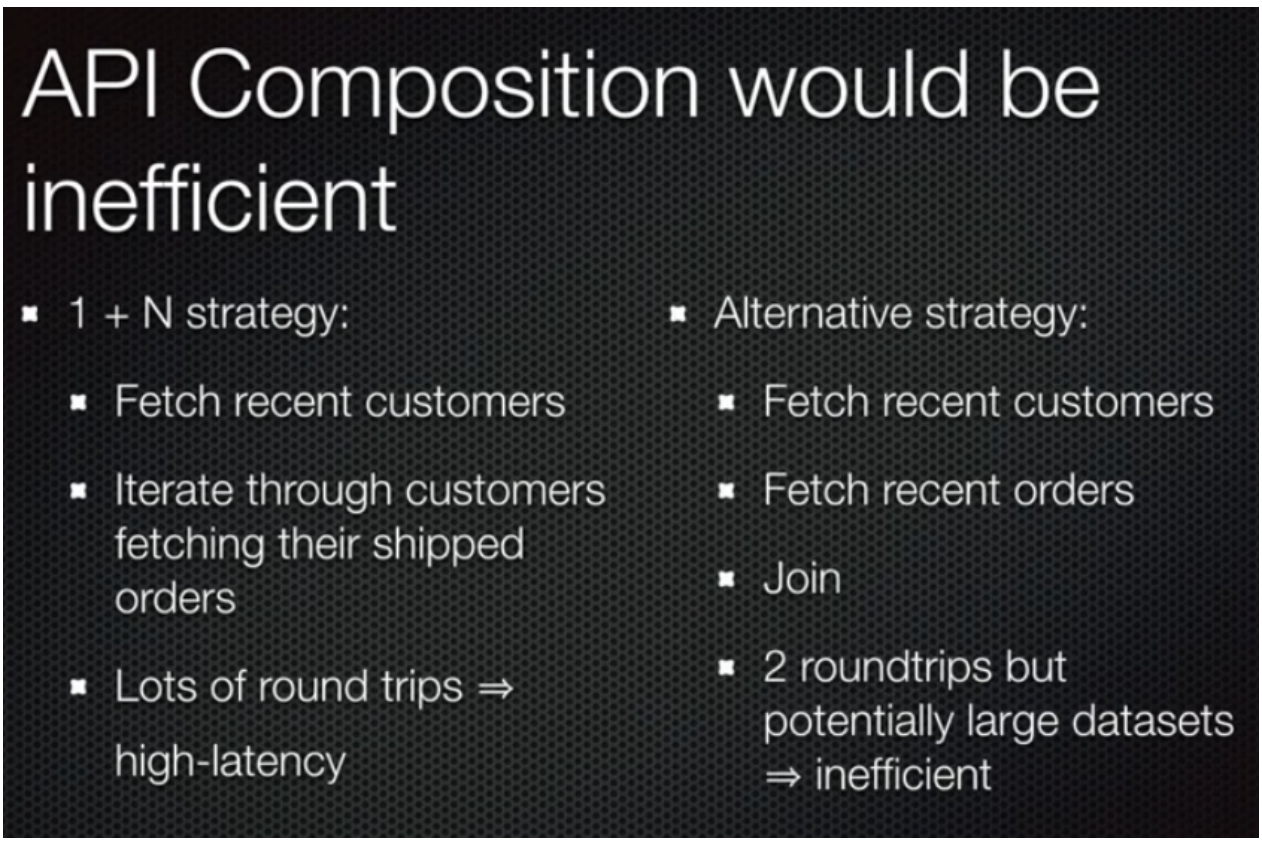


Image source

- API decomposition pattern - Gateway splits data from a single request among multiple services and gathers response. That is because each service needs only part of its data. *Atomicity* has to be insured for such cases because if we encounter an issue in later stages we need to revert all the previous ones. 27 30

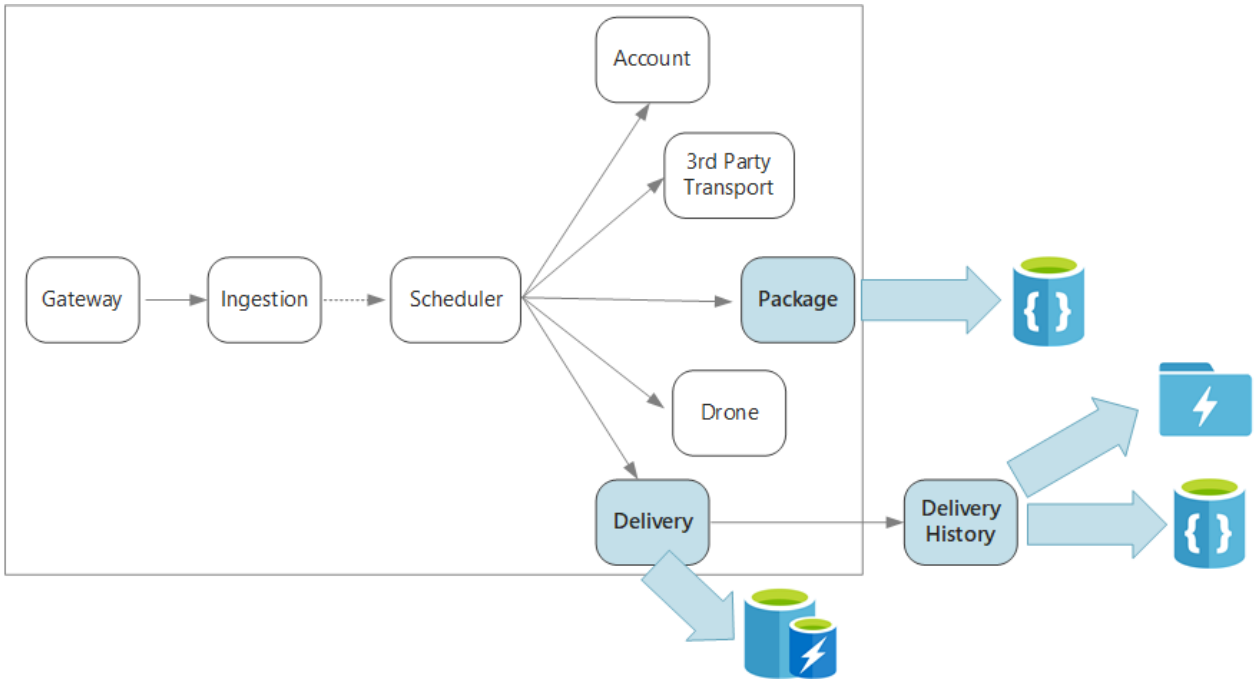


Image source

- CQRS - Technique that separates read and write models in order to provide better read models for otherwise computationally intensive use cases. It's different from Analytical storage (aka OLAP, Warehouse) in the sense of the way it's used but is similar in the way that it's also a projection (cache, view) of data which is used to provide aggregated data, but to the application itself, not for analysts

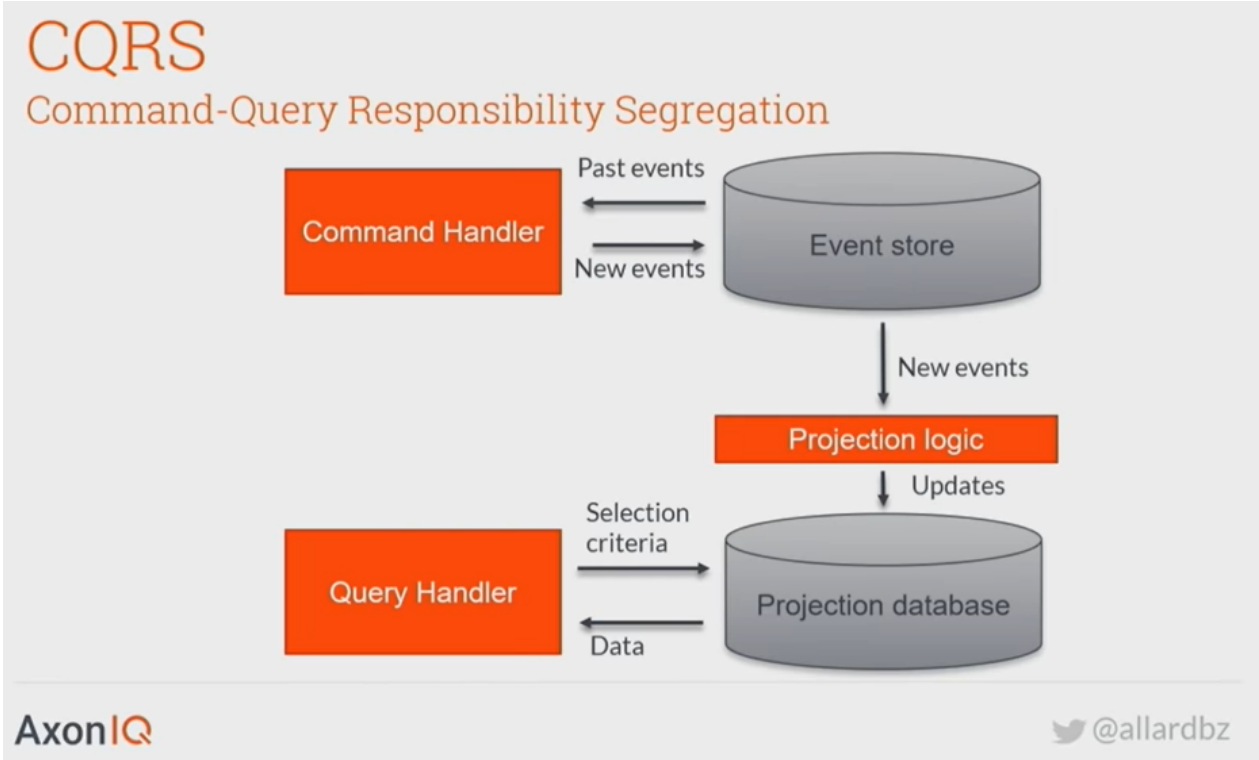


Image source

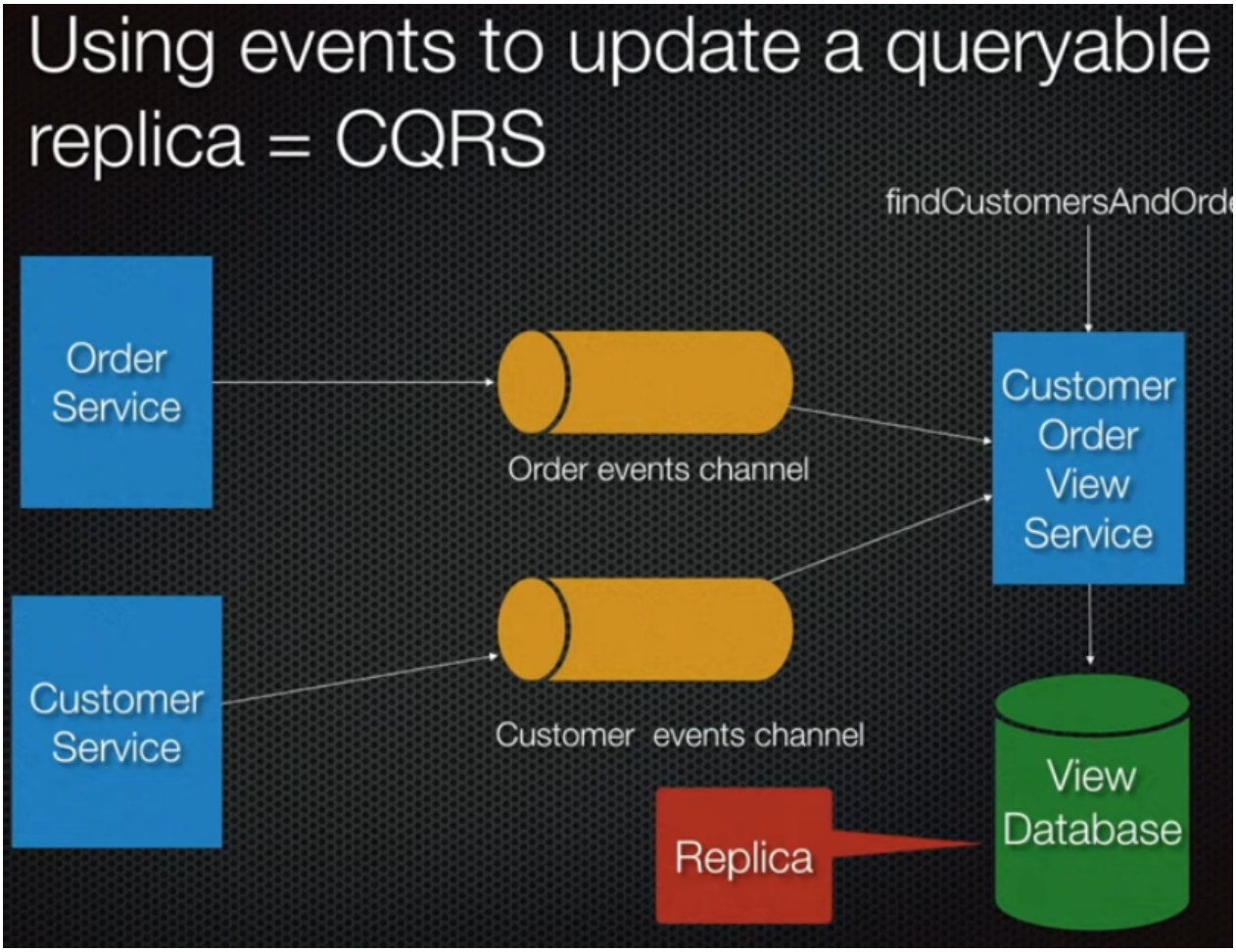


Image source

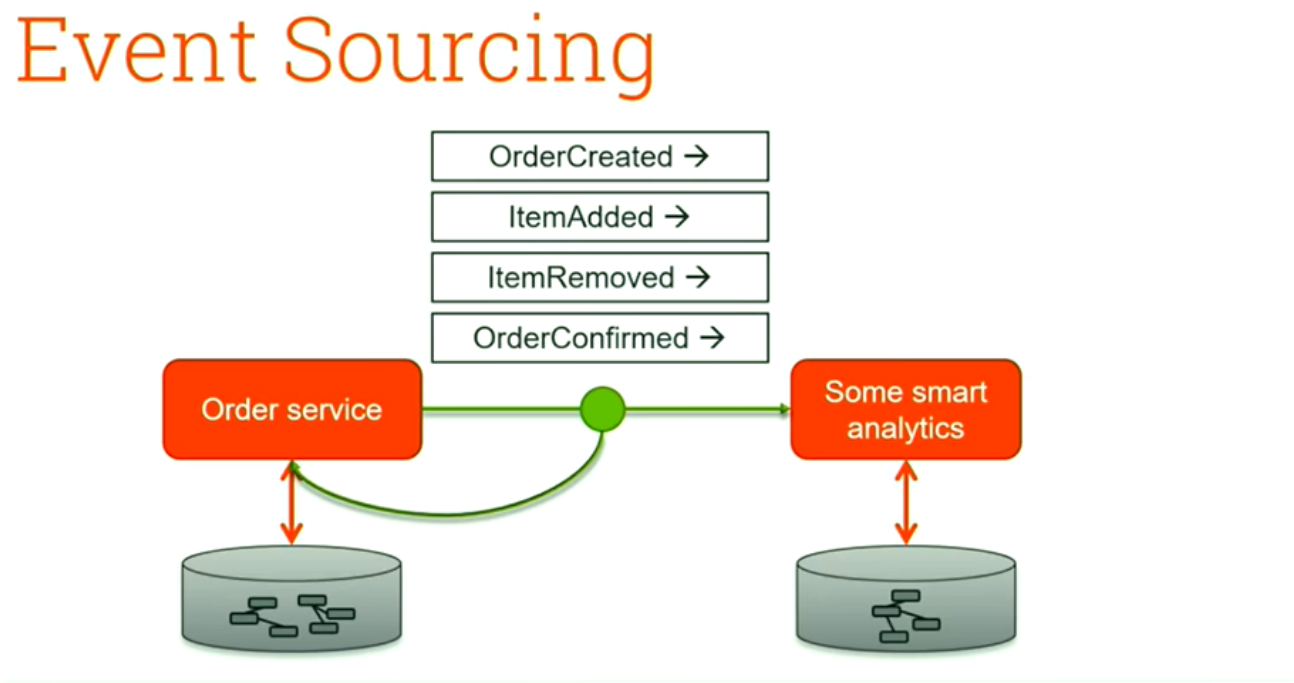
Shared DB

Use a single DB for multiple services. It's only advisable in a single bounded context because it's hard to track what has to be changed across different microservices unless DB schema is stored as a code and is automatically checked. Also, it's hard to find responsible for issues service, if the data is changed by 2 things, therefore:

- To simplify, allow only one microservice to write
- Schema changes coordination has to be present therefore all the services have to be within a single bounded context. Important to note that one team has to deal with all dependent services
- The way data is stored may not satisfy all the use cases, in which situation other patterns have to be used

Event sourcing

This approach is similar to the way a source control system works (remember, however, that Git stores complete files, to the diffs), we use event store as the single source of truth. It's 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. [4] [29] [31] [32]



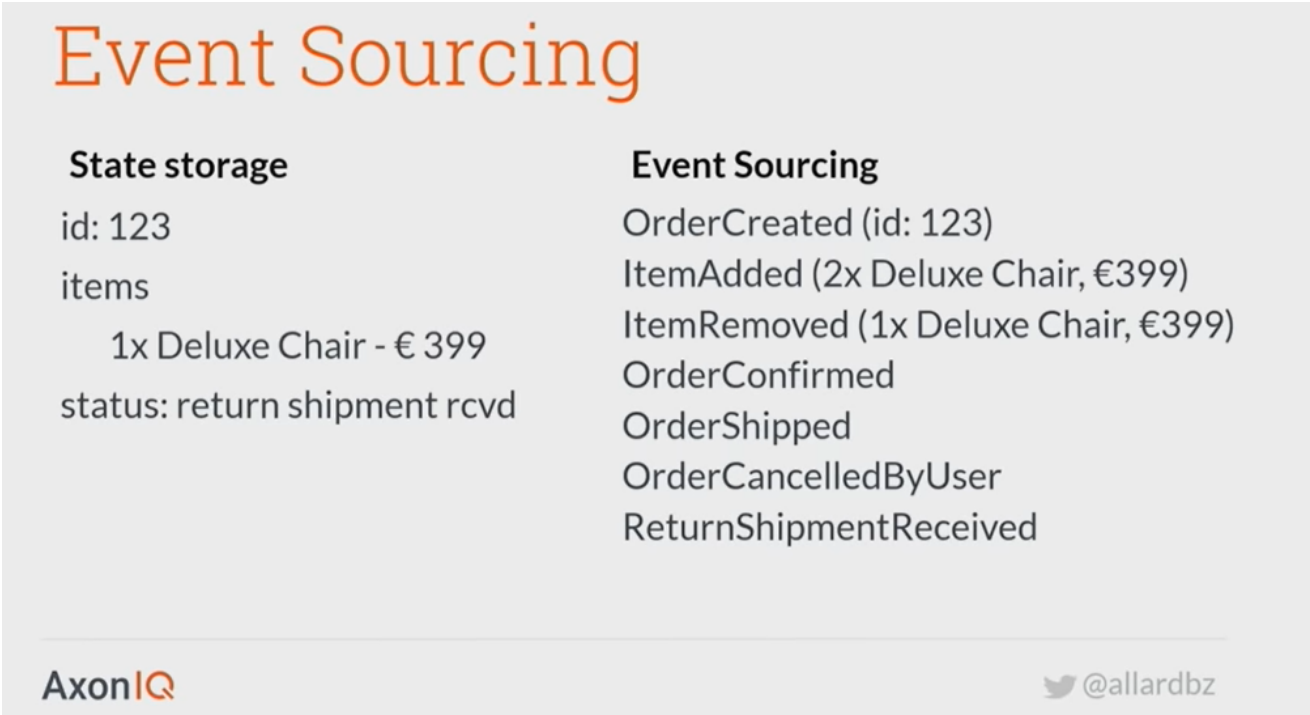


Image source

The pattern should be used to achieve the following goals: 4

Business reasons:

- Auditing, compliance, transparency - the events stream provides the possibility to endlessly check it for patterns, see individual steps that lead to the 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
- Deal with frequently changing requirements or unknown future requirements Netflix case

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 previous one.
- Single source of truth - events store now contains all the information needed for a new service to be able to receive needed information.
- Concurrency, conflict resolution - stream of events is easier to parallel and resolve conflicts if such occur because you have the history.
- Facilitates debugging - you have a sequence of events, just 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.
- Easily capture intent - it's easier to understand what an application is doing when you have a sequence of events compared to logic that is scattered over many microservices.
- Deal with complexity in models - event sourcing forces us to follow narrow-focused models that are easier to comprehend.

You'd only choose the pattern if a business reason is met and must understand the peculiarities you take on: 33

Pros:

- Debug - go back in time and see what happened, reproduce.
- History - going through history like with version control.
- Alternative state - in the case of an issue, events are possible to change and rebuild all the states.
- Memory image - all changes can be operated in memory without persisting, which improves speed

Cons:

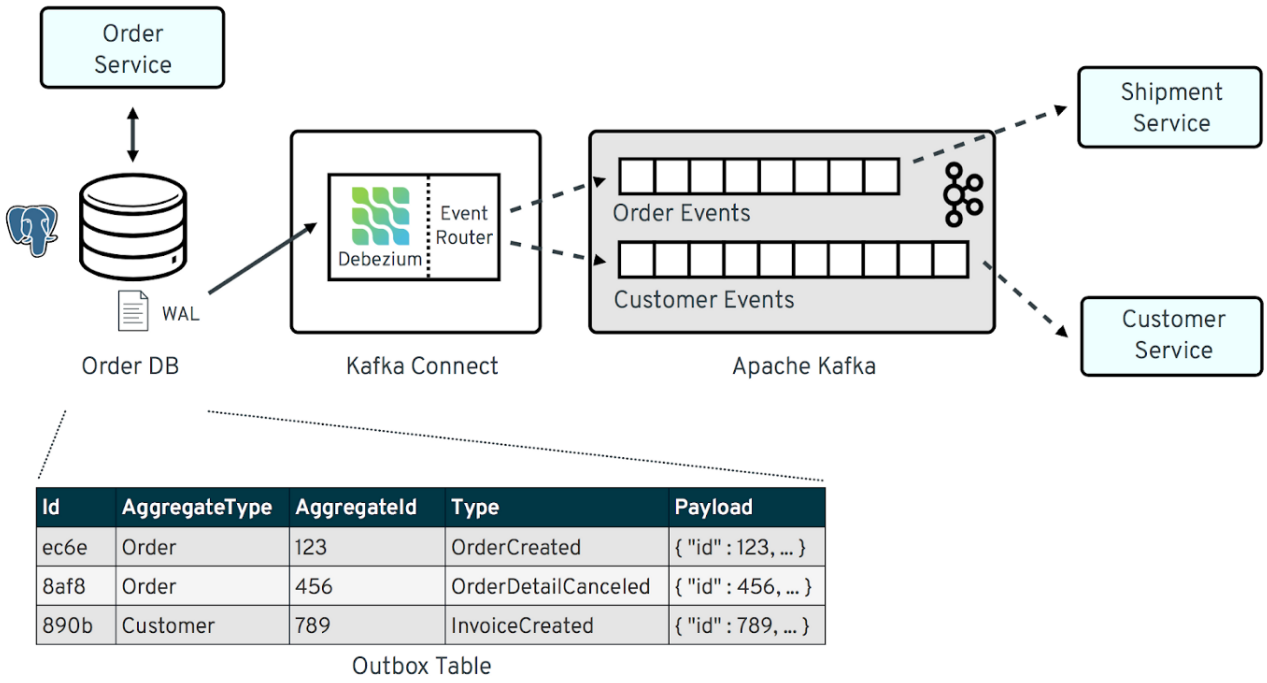
- ~~Big loads of data~~ - not a problem, storage is cheap + solved with snapshotting which can be compared to the term "closing" in accounting, when at the end of a year all expenses and incomes are calculated and basically snapshot is made to bade or future transactions upon.
- ~~Complex~~ - not a problem as everything is complex at the start. You aren't forced to rewrite the whole system, only use the pattern where you need flexibility.
- Event thinking - wrap your head, forget about the state to only think in terms of a sequence of events. Event storming is a great technique to decompose a process into a number of events.
- With many events, it's hard to materialize entity - CQRS is used to cache entities

Change data capture

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

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 as

well as the approach is not as easy. In the case of CDC, we usually operate only with dummy events of two types: upsert and delete, moreover, we do not opt into the single source of truth scenario by using techniques like DB transaction log tailing or other ones, therefore keeping the DB as the main source.



Id	AggregateType	AggregateId	Type	Payload
ec6e	Order	123	OrderCreated	{ "id" : 123, ... }
8af8	Order	456	OrderDetailCanceled	{ "id" : 456, ... }
890b	Customer	789	InvoiceCreated	{ "id" : 789, ... }

Outbox Table

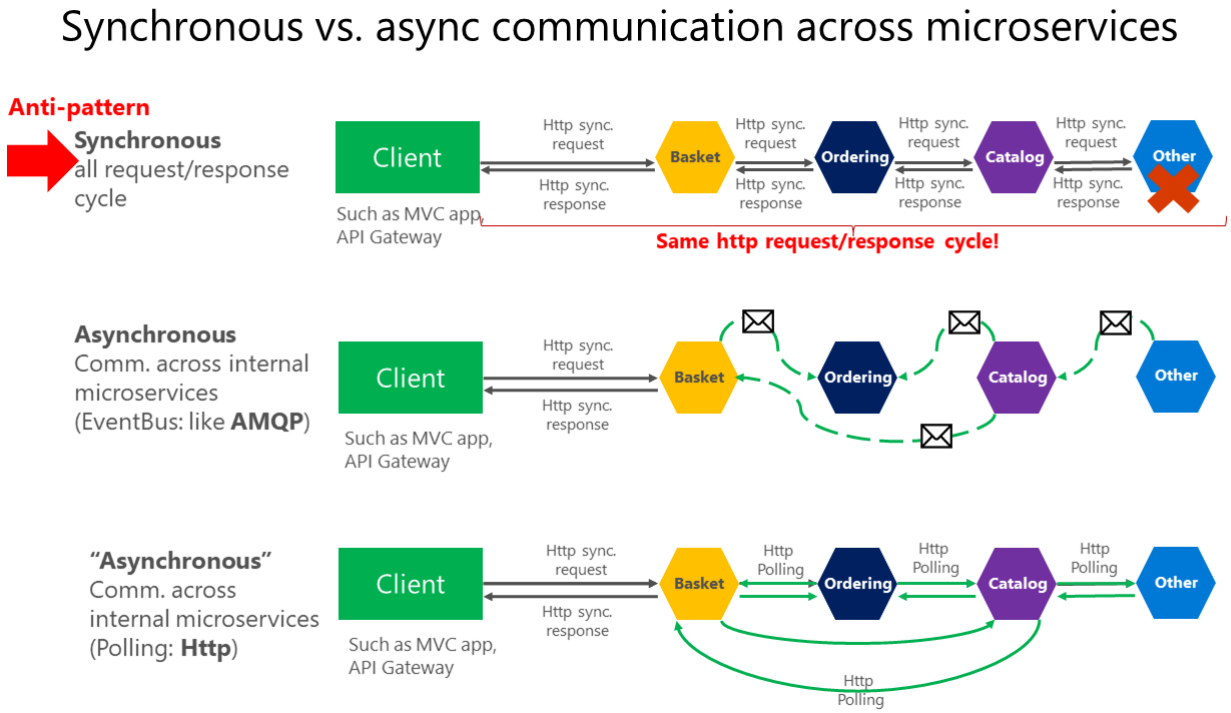
Image source

Communication

The next important bit apart from what to share is how to do it. Communication patterns are responsible for the way we build connections between components and generally are chosen based on a number of factors:

Strategy

The communication strategy can be either synchronous or asynchronous, which is different from the protocol like HTTP or AMQP, it's rather a concept related to communication when we do not call other microservices 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 microservices from client applications or from API Gateways, that's fine. If you create long chains of synchronous HTTP calls across microservices, communicating across their boundaries as if the microservices were objects in a monolithic application, your application will eventually run into problems with performance, coupling and failure propagation". 35



[Image source]

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, if we care about the condition being true later on after the initial request, basically we want to reserve some resources, and therefore strive to achieve atomicity in a distributed system. The same goes for a series of mutations that have to either succeed all or none.

Atomicity in a distributed system

Atomicity is first of all driven by the domain. Do we need the operation to be atomic? Do we care about reverting changes if when multiple services are called we fail in the middle and some data has already been affected by those other services? Do we care about the reversion of changes when we update two different documents in NoSQL 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.
- Single document write 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 doesn't mean that it isn't possible to atomically update multiple documents, it's just that a different approach is required. [36]
- 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. 29 37

So, when we talk about different microservices, there is no consistency between them unless we utilize 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's probably the first approach that comes to mind when operation atomicity has to be insured across multiple microservices: let's 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 during the second "commit" phase, which, again, can be otherwise leading to partially committing transactions in a few services but not all.

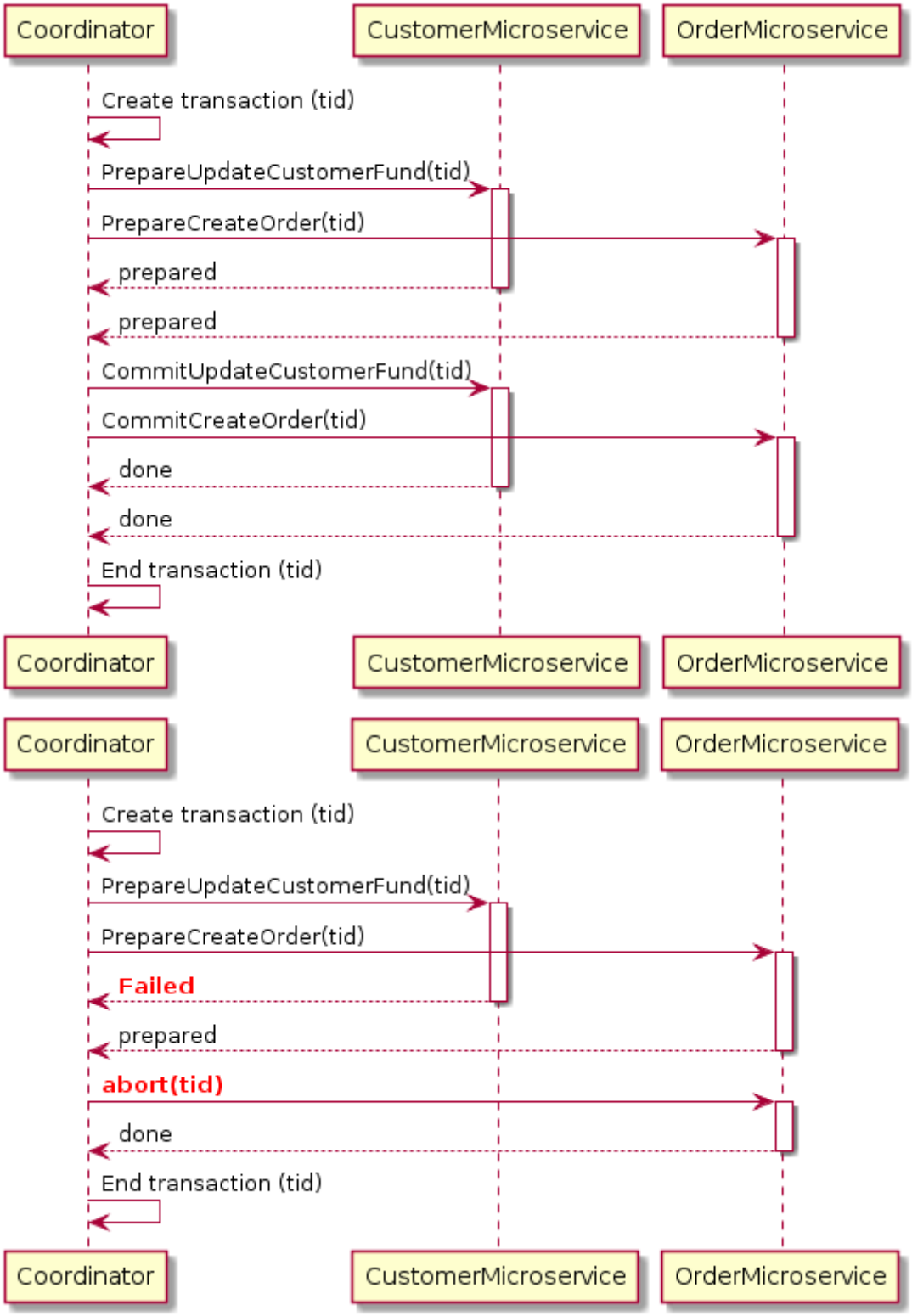


Image source

Saga pattern

Compared to the previous approach, instead of fitting everything into the same transaction boundary, we accept eventual consistency. The idea is to break the whole process into stages that are triggered or reverted using a message. In case of failure of one of the stages, all others have to be reverted by listening for and handling a compensating event. 29 39 40

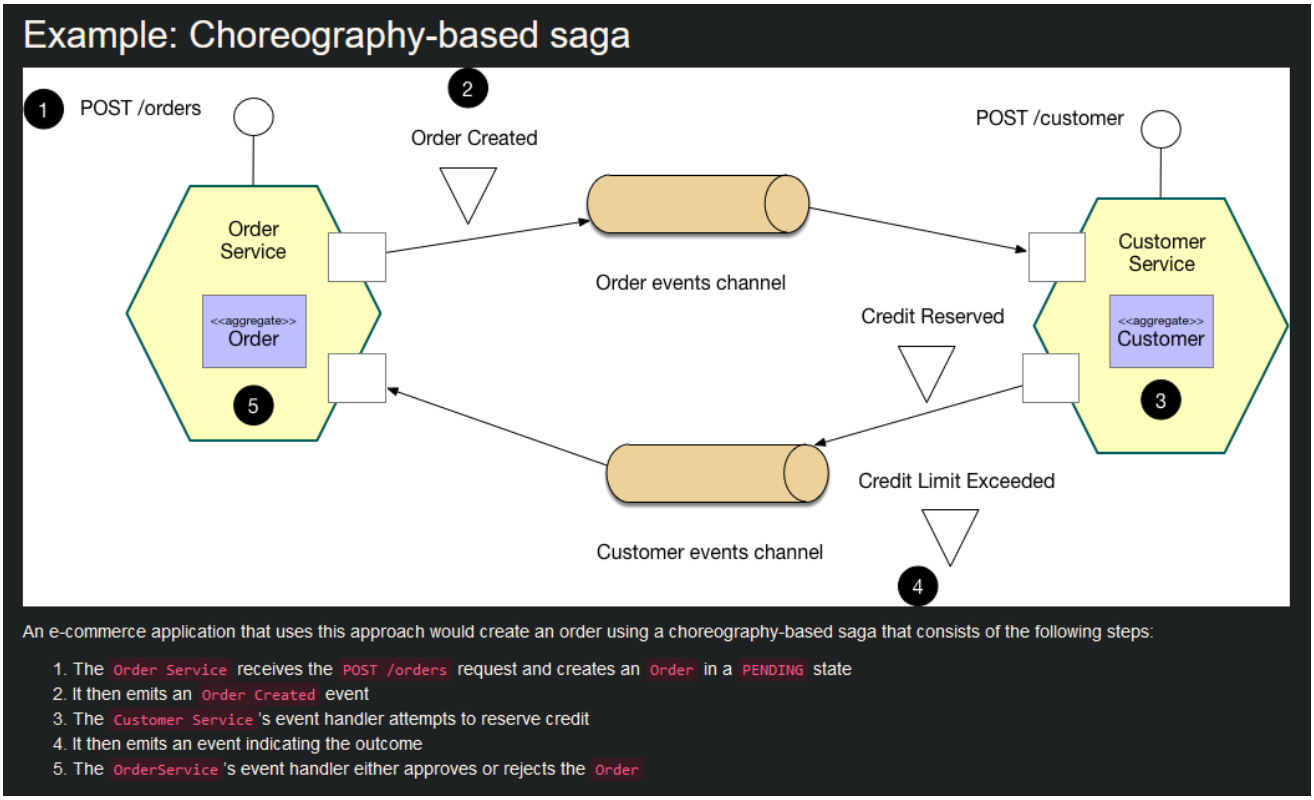


Image source

When working with Sagas, observability is a must due to the complexity of communication between many components. Another culprit is the need to implement and test compensating operations which becomes not that easy. Until a saga is committed, the data can possibly 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 fore is all the completed transactions and later 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's not like one is superior to another, which is a very 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's to call, how many components we talk to 41

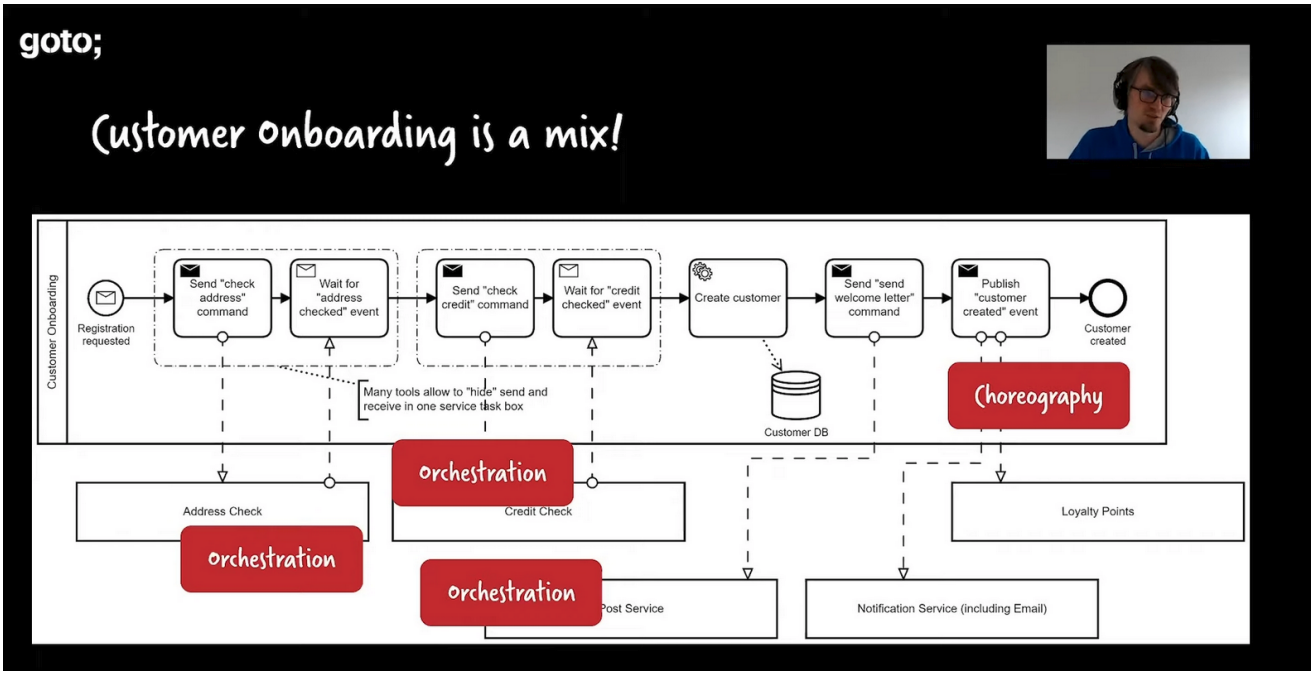


Image source

Consistency

There are cases, when we need the data to be precise, when strong consistency is needed, in this case, either the microservices have to be combined into one or a synchronous call is used. As it's been said, however, in most cases we tolerate eventual consistency for data - the user may have been downgraded, but it's 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 - WEBHOOKS, RSS with sane timeouts, and special care. 42

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 with the intention 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 indicate something that has happened when the publisher doesn't 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 has to 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 producer of the event to the consumer due to the fact that the consumer has to 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. [4] [43]

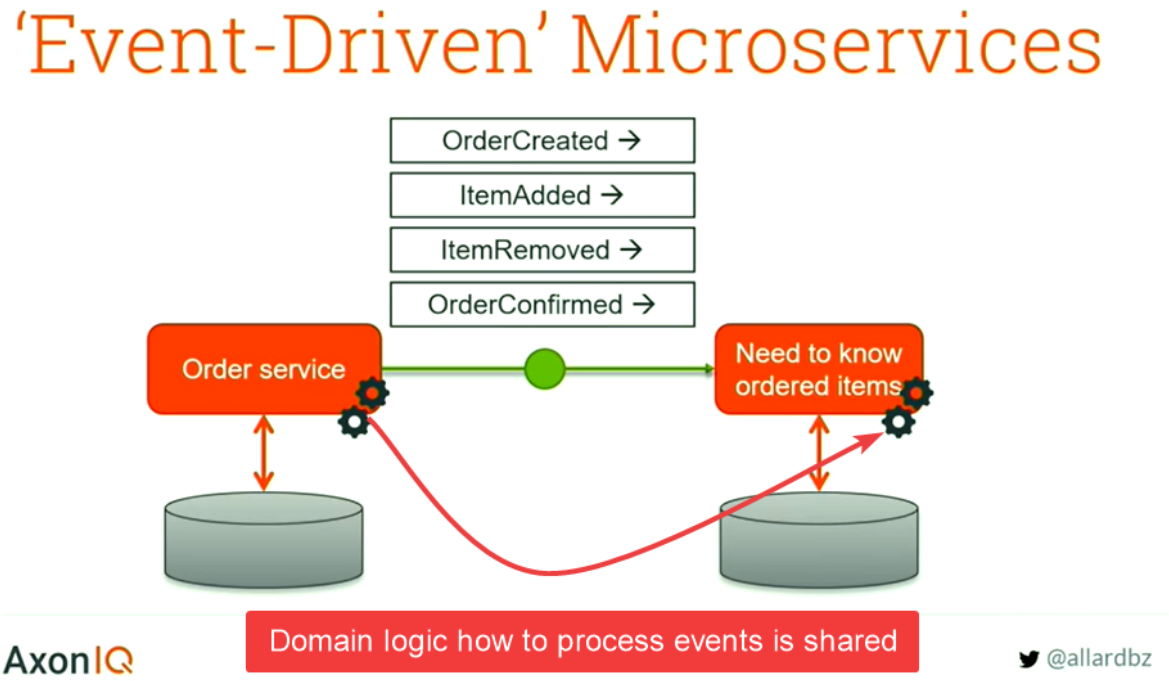


Image source

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 have to be a single one and prevent the chattiness. What changes together, goes together.

With microservices, we want to ensure as minimal as possible interface. Therefore, we should think about what we want to share and create an anti-corruption layer to prevent exposure of private, internal for this service knowledge. [10]

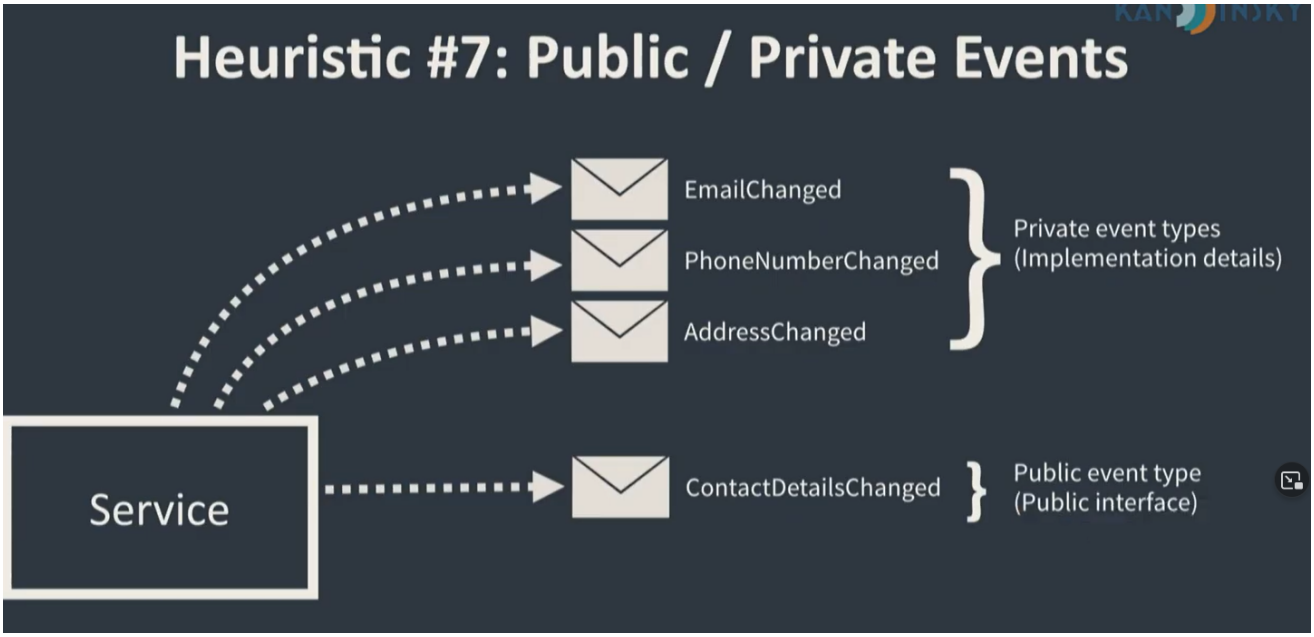


Image source

By exposing domain events, we make the consumer understand and probably process the event, which isn't always something we want. In case only one other service understands the language (one process stopped here, another starts there), we may permit it, but if there are a couple of such services, that may pose a problem. [4 10]

Image source

For a consumer, it's not always clear what events mean. Therefore it's important to make them more intelligible, removing ambiguity. In the example, it's hard to understand whether we need to overwrite or skip the assignment

Image source

Types of events

To give more insights, there are different event types depending on the payload: 44

- Notification event (aka shallow event) - contains only ids; the less data event contains the fewer efforts it's required to maintain schema. The downside is the service publishing the event will be bombarded with calls for more data.
- Event-carried state transfer - contains 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 bounded contexts but utilize an anti-corruption layer to map events to more specific with appropriate granularity and payload for different bounded contexts.

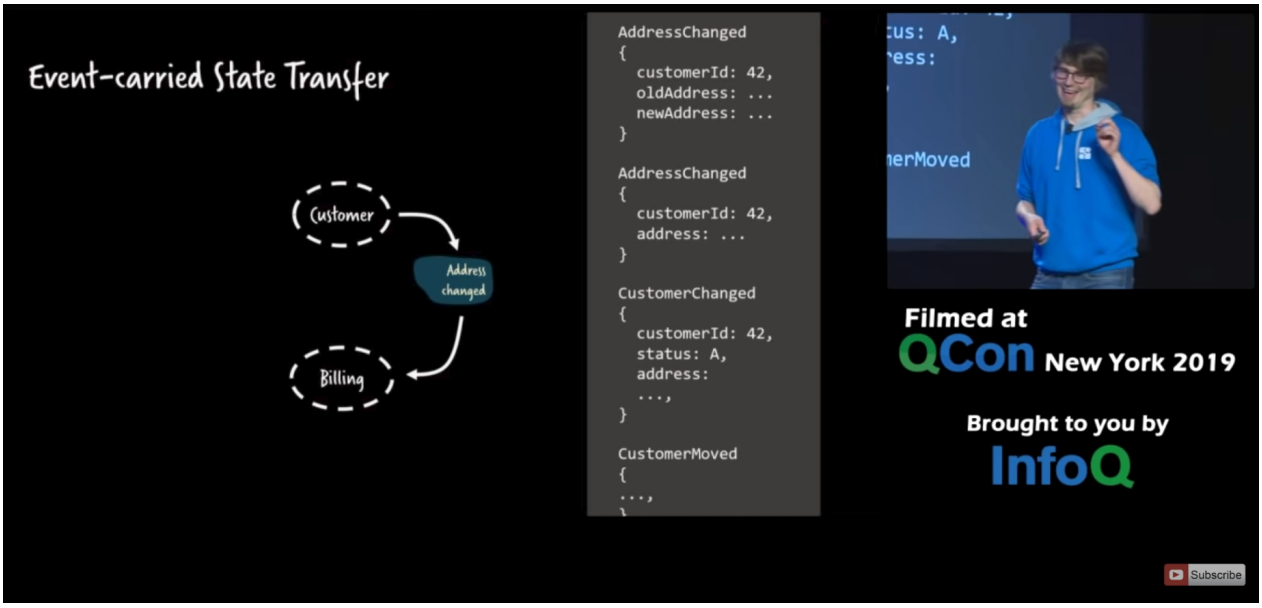


Image source

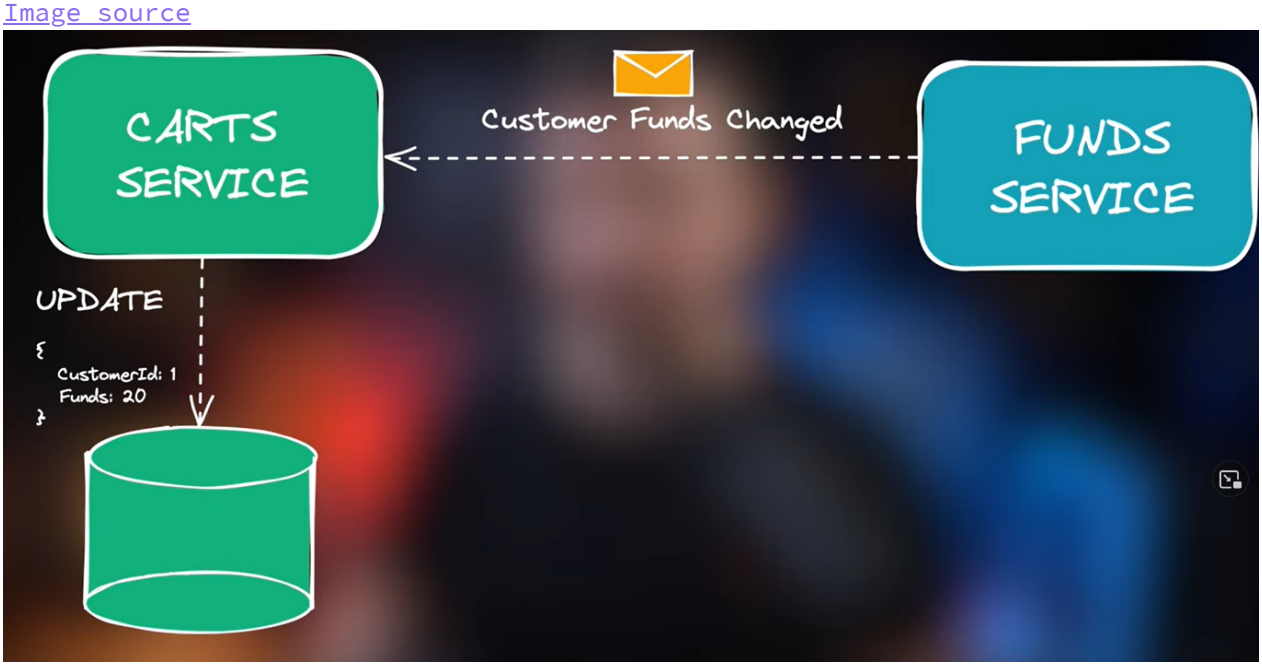


Image source

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

- Domain-specific - have fewer data and more meaning. Have to be used within a single context and carefully propagated outside, most likely 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 (relative to other actions).

The 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's hard to achieve in some systems like RabbitMq, where, if compared with Kafka, there is no native support for partitions.

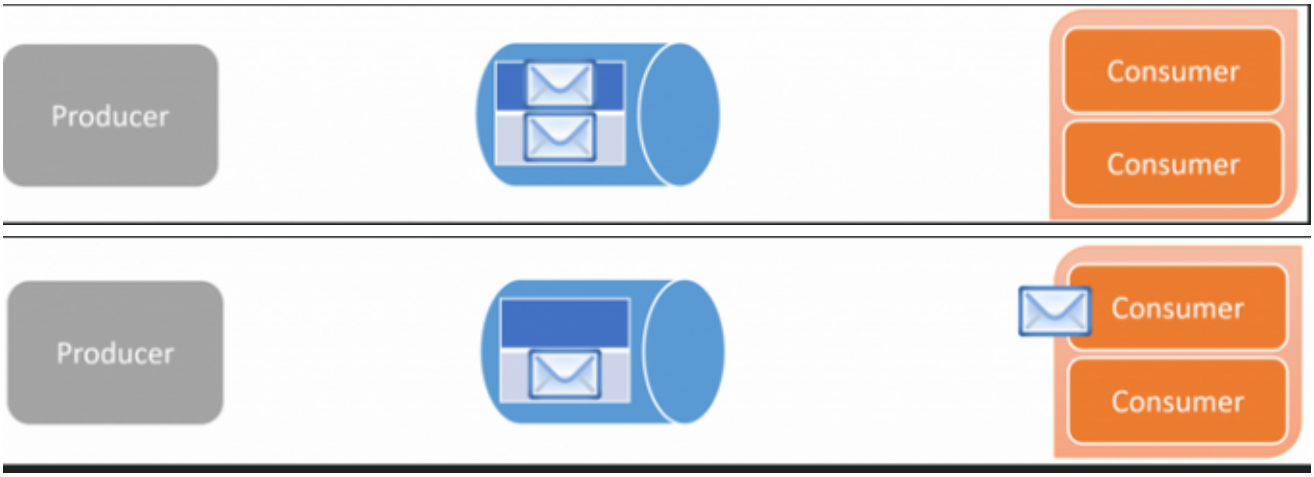


Image source

The rule of thumb is to develop a system so that it doesn't depend on message ordering. For example, publish the next event only after the previous was handled and an acknowledgment was sent (can be seen in the Saga pattern) or take into account cases when events are out of order. For example, an order can be canceled before it's Accepted. Usually, it's 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 has to be changed when the schema does so. Events have the schema as well, the difference is that now the publisher doesn't 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. 44

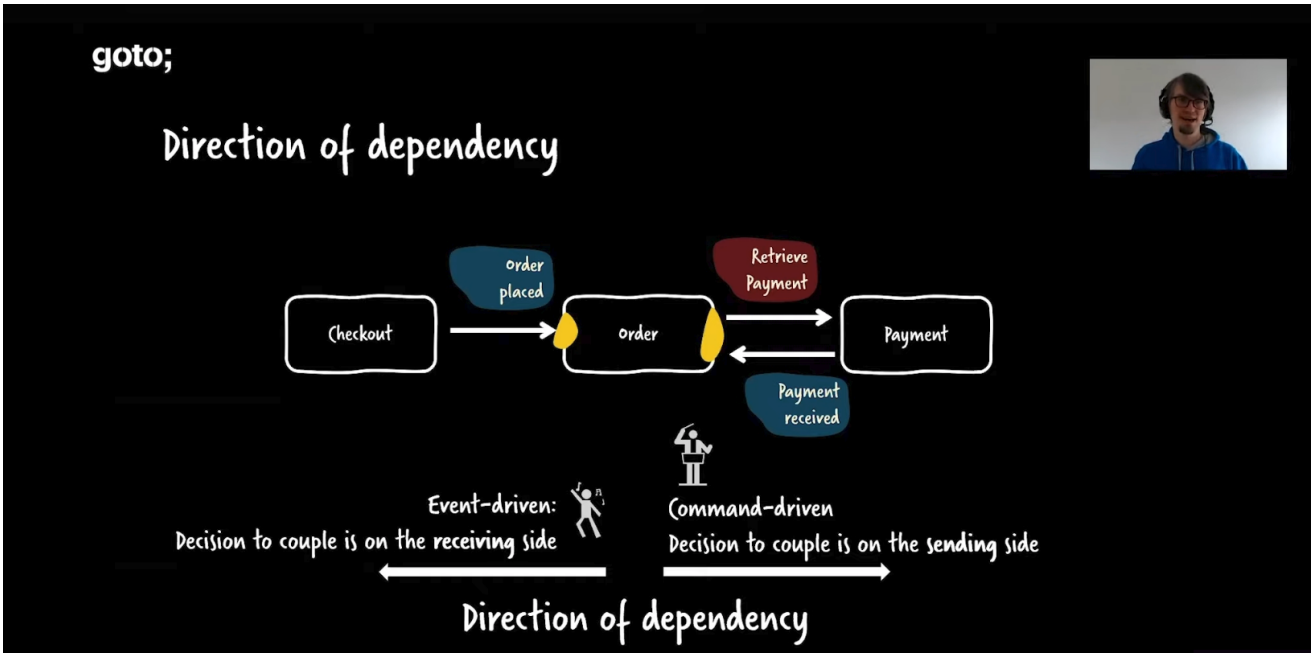


Image source

This is the same as the dependency inversion principle in the layered architecture, we use it to move the responsibility to the appropriate side.

In the example below, auth service is listening to events, therefore dependent. When we change any of the services, we also need to change the auth service.

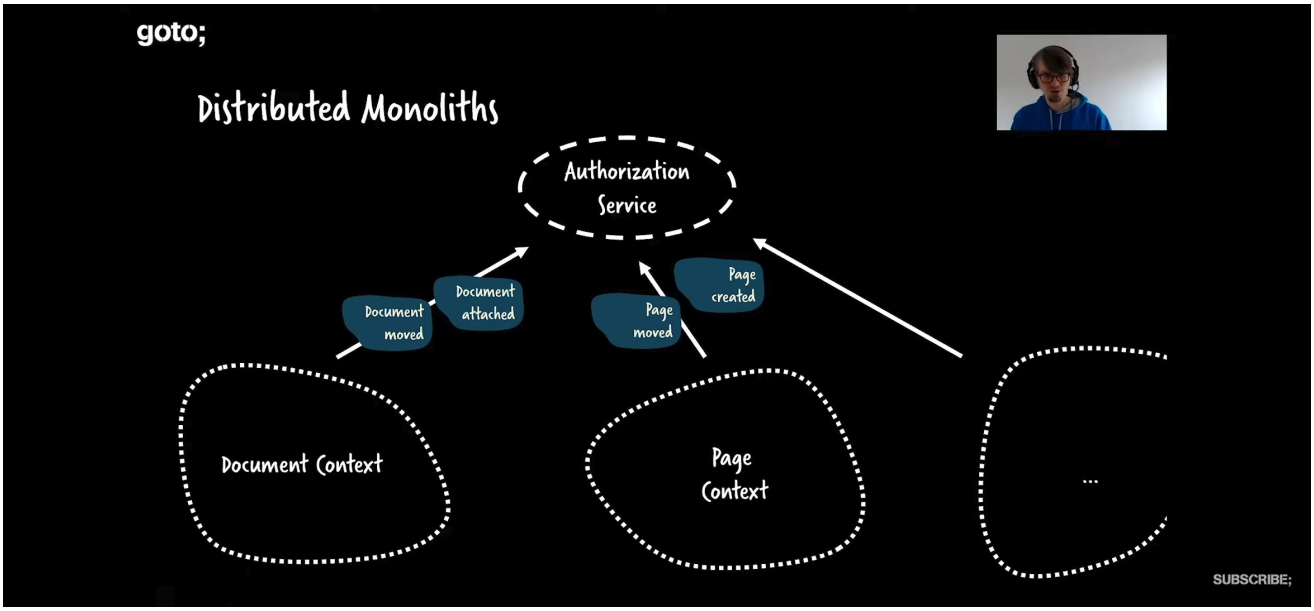


Image source

If we use a command, we inverse the dependency and now we do not need to change auth service, but vice versa. Auth service is considered to be less frequently changed.

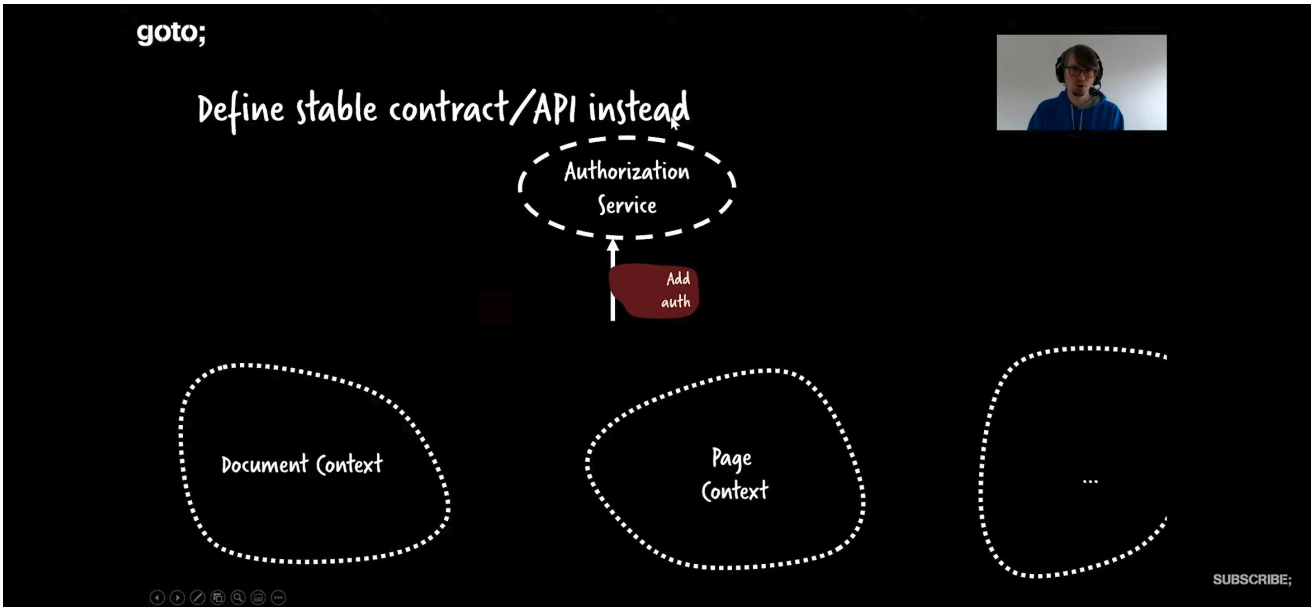


Image source

Idempotency for commands and events

Due to how distributed systems 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 can't 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, it doesn't matter how many times we run the operation
- save id of handled requests and check if it isn't already handled (idempotency key)
- other ways

Airbnb implemented **"Orpheus", a general-purpose idempotency library**, across multiple payments services with an **idempotency key** which is passed into the framework, representing a single idempotent request. Paypal implemented idempotency in the API using MsgSubId (Message submission ID) and Google Service Payment implemented idempotency with request ID.

Conclusion

This talk, as with models in DDD, is just a project of highly interconnected learned material, which, hopefully, helped you understand that designing a reliable distributed system isn't that easy and has to be approached consciously. The nature of distributed systems differs dramatically from what we used to think when dealing with monoliths. When compared, even though both monoliths and microservices are built to satisfy the business need, the technical goals that are achieved vary a lot. So, for the microservices 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 is not to jump on the bandwagon of microservices, unless to the learning 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 Docker, k8s, Event sourcing, etc.

So, 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 times. Various paradigms are developed to solve problems that follow a common pattern in an efficient way, microservices is one of such ways.

Developing microservices, we strive to minimize the dependency between units that is directly influenced by the structure of the organization 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 between boundaries, we share data, which is the prime contributor to the increase of coupling in a system, consequently, we have to approach the 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 have to be noted.

> "Risk comes from not knowing what you're doing." - Warren Buffett
> "I don't pretend we have all the answers. But the questions are certainly worth thinking about." - Arthur C. Clarke

What is next

Covering a wide surface related to the development of distributed systems, this article didn't dive into many details, referencing external sources. Apart from those used during the preparation of the material and referenced at the end, there are books that I haven't read completely, but were suggested commonly:

- Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems - Reading. Data - storing, formats, consistency.
- Building Microservices: Designing Fine-Grained Systems - Haven't read. Many of the covered today's topics. Also, this video is a short overview of his book.
- Accelerate - Haven't read. Teams communications.

- Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions - Haven't read.
- Learning Domain-Driven Design - Haven't read. From the content table - many of the covered today topics.

> There are a couple of relative topics that are vital to developing complex distributed system. Although bounded contexts increase the problems locality, still in distributed system we need absolute observability. Another problem is depenencies. When there are many parts in a system, it's hard to track dependencies by using diagrams, we need to know what external components are going to be influenced and test them. These are topics I'm working on, but the next time I'll descend a level lower and will speak about about

References

1: https://vladikk.com/2018/02/28/microservices/ "Tackling Complexity in Microservices"
2: https://www.youtube.com/watch?v=0TYbHVc2yWI "Autonomous microservices don't share data. Period - Dennis van der Stelt"
3: https://youtu.be/lE6Hxz4yomA "Eric Evans: What I've learned about DDD since the book"
4: https://www.youtube.com/watch?v=DzGuDNHsOQ0 "Event-Driven Microservices - not (just) about Events! • Allard Buijze • GOTO 2018"
5: https://en.wikipedia.org/wiki/Coupling_(computer_programming) "Coupling (computer programming)"
6: https://www.infoq.com/news/2020/02/balancing-coupling-ddd-europe/ "Balancing Coupling in Distributed Systems: Vladik Khononov at DDD Europe"
7: https://vladikk.com/2020/04/09/untangling-microservices/ "Untangling Microservices, or Balancing Complexity in Distributed Systems"
8: https://learn.particular.net/courses/distributed-systems-design-fundamentals-online "Distributed Systems Design Fundamentals"
9: https://www.youtube.com/watch?v=jdliXz70NtM "Finding your service boundaries - a practical guide - Adam Ralph"
10: https://www.youtube.com/watch?v=dlnu5pSsg7k "Bounded Contexts, Microservices, and Everything In Between - Vladik Khononov - KanDDDinsky 2018"
11: https://www.youtube.com/watch?v=zkRfDw0N4W8 "Dissecting Bounded Contexts - Nick Tune - DDD Europe 2020"
12: https://www.youtube.com/watch?v=am-HXycfalo "Bounded Contexts - Eric Evans - DDD Europe 2020"
13: https://learn.microsoft.com/en-us/dotnet/architecture/microservices/architect-microservice-container-applications/logical-versus-physical-architecture "Logical architecture versus physical architecture"
14: https://vladikk.com/2018/01/21/bounded-contexts-vs-microservices/ "Bounded Contexts are NOT Microservices"
15: https://www.youtube.com/watch?v=ez9GWESKG4I "The Art of Discovering Bounded Contexts by Nick Tune"
16: https://en.wikipedia.org/wiki/Conway%27s_law "Conway's law"
17: https://learn.microsoft.com/en-us/dotnet/architecture/microservices/architect-microservice-container-applications/identify-microservice-domain-model-boundaries "Identify domain-model boundaries for each microservice"
18: https://vladikk.com/2018/01/26/revisiting-the-basics-of-ddd/ "Revisiting the Basics of Domain-Driven Design"
19: https://www.youtube.com/watch?v=sFCgXH7DwxM "DDD and Microservices: At Last,

Some Boundaries!"
20: https://microservices.io/patterns/refactoring/strangler-application.html "Pattern: Strangler application"
21: https://learn.microsoft.com/en-us/azure/architecture/patterns/strangler-fig "Strangler Fig pattern"
22: https://www.youtube.com/watch?v=Ab5-ebHja3o "Practical DDD: Bounded Contexts + Events = Microservices"
23: https://www.youtube.com/watch?v=3-_0ZcI2SYc "Integrating Bounded Contexts - Indu Alagarsamy - DDD Europe 2020"
24: https://www.youtube.com/watch?v=Y1ykXnl6r7s "Find Context Boundaries with Domain Storytelling - Stefan Hofer and Henning Schwenter - DDDEU 18"
25: https://en.wikipedia.org/wiki/CAP_theorem "CAP theorem"
26: https://www.youtube.com/watch?v=w9GP7MNbaRc "Why Distributed Systems Are Hard"
27: https://www.youtube.com/watch?v=hev65ozmYPI "All our aggregates are wrong - Mauro Servienti"
28: https://auth0.com/blog/introduction-to-microservices-part-4-dependencies/ "Intro to Microservices, Part 4: Dependencies and Data Sharing"
29: https://www.youtube.com/watch?v=kyNL7yCvQQc "Not Just Events: Developing Asynchronous Microservices • Chris Richardson • GOTO 2019"
30: https://learn.microsoft.com/en-us/azure/architecture/microservices/design/data-considerations "Data considerations for microservices"
31: https://www.youtube.com/watch?v=8JKjvY4etTY "Event Sourcing • Greg Young • GOTO 2014"
32: https://www.youtube.com/watch?v=Hlb-Ss3q3as "Building Streaming Microservices with Apache Kafka - Tim Berglund"
33: https://www.youtube.com/watch?v=osk0ZBdBbx4&list=PLJRhuqXHXswxDLwneCeLtbSYVkyhCa4Ve&index=40 "Mistakes made adopting event sourcing (and how we recovered) - Nat Pryce - DDD Europe 2020"
34: https://www.youtube.com/watch?v=Lqv-q2VWNQM "Practical Change Data Streaming Use Cases with Apache Kafka & Debezium"
35: https://learn.microsoft.com/en-us/dotnet/architecture/microservices/architect-microservice-container-applications/communication-in-microservice-architecture "Communication in a microservice architecture"
36: https://jimmybogard.com/life-beyond-distributed-transactions-an-apostates-implementation-aggregate-coordination/ "Life Beyond Distributed Transactions: An Apostate's Implementation - A Primer"
37: https://microservices.io/patterns/data/transactional-outbox.html "Pattern: Transactional outbox"
38: https://developers.redhat.com/blog/2018/10/01/patterns-for-distributed-transactions-within-a-microservices-architecture "Patterns for distributed transactions within a microservices architecture"
39: https://microservices.io/patterns/data/saga.html "Pattern: Saga"
40: https://learn.microsoft.com/en-us/azure/architecture/reference-architectures/saga/saga "Saga distributed transactions pattern"
41: https://www.youtube.com/watch?v=zt9DFMkjkEA "Balancing Choreography and Orchestration • Bernd Rücker • GOTO 2020"
42: https://www.youtube.com/watch?v=eW4JgrkwWEM "Microservices communication patterns, messaging basics, RabbitMQ | Messaging in distributed systems"
43: https://www.youtube.com/watch?v=STKCRSUsyP0 "The Many Meanings of Event-Driven Architecture • Martin Fowler • GOTO 2017"
44: https://www.youtube.com/watch?v=jjYAZ0DPLNM "Opportunities and Pitfalls of Event-driven Utopia"

45: https://codeopinion.com/message-ordering-in-pub-sub-or-queues/ "Message Ordering in Pub/Sub or Queues"