

**What is right sizing?**

If you have experience working in Agile and Scrum teams, we will size our user stories based upon t

shirt sizing, based upon how complex the user story is, we are going to assign one of the t shirt size for the user story.

The same applies for the microservices as well.

is about how to right size and identify service boundaries of microservices.

When I say right sizing, I mean your microservices should not be so big where you are not able to grab all the advantages that we have with the microservice and at the same time they should not be so small and they should not have very less business logic due to which we will end up having more number of microservices inside the organization.

If you onboard more number of microservices which are very small in nature, then definitely there will be some operational overhead to make sure all these microservices are interconnected.

They are able to communicate with each other. There's going to be lot of operational overhead.

That's why as an architect, as a developer or as a technical lead or as a manager, it's our responsibility to make sure our microservices are following the right sizing based upon the business needs.

The very next question that you may have here is I understand what is right sizing, how to derive the

sizing for my own microservices inside my own project.

For the same, there are two most followed approaches or standards in the industry.

**The very first approach is Domain-Driven-Sizing.**

Since most of our business logic are enhancements driven by the business needs, we can size our define the boundaries for our microservice that are closely aligned with the domain driven and business capabilities.

When I say domain driven, for example, inside the EasyBankApplication, we have various domains

like Accounts, Cards, Loans, and very similarly, there can be other domains inside the easy bank.

In other words, we can call these domains as departments or different verticals inside an organization.

We can build microservices based upon these domains.

For example, we have built three different microservices by taking these domains as an input like accounts, cards and loans.

But there might be scenarios where an organization can have a big domain and a single domain,

there might be a lot of products being handled or a lot of business being handled.

And under that domain there will be hundreds of developers.

In such scenarios, it may not be a good idea to go strictly with Domain-Driven-Sizing.

That's why in such scenarios we should not blindly go based upon the domains or business capability

departments or verticals inside an organization.

We should discuss with all the business people, technical people, architects inside the organization

who has the domain knowledge.

There might be some people who might be working in the organization from many decades.

We should take inputs from all those leaders and experienced people to understand what a domain is doing on day to day operations, because no single person will never know all the business happening inside the organization.

That's why we need to talk with the domain experts and talking with these domain experts, business

leaders, along with the technical people, clients, business analysts, product owners.

So this is going to be a super time consuming process.

Definitely,

It is not going to happen in a 1 or 2 days. At least,

It will need 3 to 6 months of time where you will gather all the domain related information.

What are the operations that are being handled by each domain?

What is the team size?

What are the existing applications?

Once we have all this information, there will be a good brainstorming happen and inside these brainstorming sessions you will involve all the business people, technical people, business people, client people, and accordingly, a right sizing will be identified for the microservice.

When you start with a size for microservice, it is obviously not going to be a right size for many

organizations.

Initially, they will start with some assumptions and with some sizing of the microservice.

Once they get started with the microservice development and deployments, if they see a lot of operational overhead is required, that means they might have identified a lot, many microservices, which is a operational overhead in such scenarios, then definitely they will revisit their sizing and they will try to club some microservices to a larger microservice.

This way this is going to be a continuous process where business will try to see if the current sizing

of the microservices is fine.

If not, they will try to fine tune it either by splitting the existing microservice or by clubbing

the microservices.

So this is one of the most common approach and this has its own advantages and disadvantages.

One of the primary disadvantages is it is time consuming and it also need a lot of people who has good understanding on the business and domain to overcome these challenges,

**we also have other approach, which is Event-Storming Sizing.**

In this scenario, a facilitator is going to conduct an interactive session among various stakeholders

like product owners, developers, testers, clients, business owners, business leaders.

Inside these fun sessions, everyone will be provided with some sticky notes.

On top of those sticky notes, they will be asked to think about an event that is possible inside a

business.

If you take a bank application as an example, a customer can complete the payment or he can search

for a product.

So these are the sample events.

Once all the possible events are identified by the stakeholders as a next step, they are going to identify what are the commands and reactions.

Command is a process which will initiate the event.

For example, for the completed payment event, the customer has to click on a button to initiate the

payment.

So that is going to be a command and the reaction can be the amount will be deducted from his account post the completion of the payment.

So this way, once all the events commands reactions are identified, they will be separated and segregated based upon the domain.

For example, all cards related events, commands and reactions.

I can move to the cards domain.

Similarly for Loans and Accounts.

The advantage with this approach is you don't need experts who knows the business from many days.

Anyone who is using your product on day to day basis, who is testing your product on day to day basis, they can provide their inputs.

Like this is the event that I can perform inside the application.

And since this is going to be a super interactive and fun sessions with lot many stakeholders, you

can gather a lot of information with a very few meetings.

Maybe you can conduct 5 to 6 meetings in a month and within a month you will have all the events information along with that domain mapping.

Once you have all these information, it is going to be super, super easy to right size your microservice.

You can see there is a blog from lucid chart how to perform these event storming sessions inside an any organization. We can try to refer the same for more details.

This is the blog.

It has all the details on how someone can facilitate these events.

You as a developer, you don't have to facilitate, but this is going to be a good information for you

because at some day you may end up attending these kind of events or in some interview people can ask you how you have rightsized your microservice in such scenarios.

All this information is going to be super helpful if you can scroll down

first, we need to provide the sticky notes to the stakeholders or with the help of Lucidchart, you

can make all this process digital with their website.

Once the sticky notes is provided, all the stakeholders, they will provide their inputs like what

are the possible events that can happen inside an application?

The advantage of this process is it is going to be super fast and it is going to be straightforward

and it is going to be engaging with all the stakeholders and it is going to be more effective compared

to the Domain-Driven Sizing.

If you can further scroll down on this page, you'll be seeing all the steps that a facilitator will

follow to perform this event.

Storming.

First, the facilitator will try to invite all the right people, including the clients, developers,

testers, managers, architects, business leaders post that they are going to provide unlimited modelling space, which means using sticky notes or any other approach.

You need to give a freedom to the stakeholders that they can mention any number of events.

There can be duplicates like two different people they can propose duplicate events.

That's fine because as a facilitator you should be removing all those duplicate events.

With this freedom, stakeholders will try to provide all the events that they are aware inside the application.

As a next step, we need to identify commands for each domain events.

Once the commands are identified, We should also identify the reactions that may happen because sometimes a reaction can be act as a command for the next event.

That's why identifying the events commands reactions is very important.

Once all these events commands reactions are identified, you can try to segregate them based upon that domains or based upon the business or based upon the departments.

So here you can see they have separated based upon business product creation, sales testing.

So very similarly, you can separate based upon the domains.

Once you have these domain specific events, this is going to be input for your right sizing of the

microservice.

If you are interested, please read all the details present inside this blog.

It is going to take ten minutes only.

I'm going to mention this link inside the GitHub Readme document.

With this I'm assuming your super super clear with the common approaches that we have to right size our

microservices. In the next lecture, let's try to take an example of easy bank and try to right size our

microservices.

**Sizing and Identifying boundaries with a Bank App use case**

In the previous lecture I discussed about the common approaches that we can follow whenever we are trying to right size and identify service boundaries for our microservices.

Now, inside this lecture, let's try to take an example of a BankApplication that needs to migrate

or built based upon a microservices architecture.

So whenever someone is trying to migrate or built based upon a microservices architecture, definitely first, they need to identify the appropriate sizing for the microservices.

So the CEO or the chief technology officer of the bank, he or she formed three different teams for

these analysis and he or she asked them to come up with their own sizing for the microservices.

Let's try to see what these three different teams, they try to come up based upon their understanding on the BankApplication.

They might have used the Domain-Driven Sizing or Event-Driven Sizing.

It's up to them, but let's try to see what they come up and towards the end of this lecture we will

act as a CEO or the CTO of the bank and will try to select one of the sizing for our microservices.

**Now let's see what is Team1 came up.**

The very first team, they came up with a sizing saying that saving account and trading account related business logic can be separated as a single microservice.

So this bank application, they support both saving account and trading account.

Using saving account you can save or you can store your money digitally.

And similarly, with the help of trading account, you can buy and sell the stocks mutual funds.

So they're trying to club both of the saving account and trading account into a single microservice.

And similarly, the bank also provides cards and loans related products to the end customers.

They're trying to club both these cards and loans as a single microservice.

So this is what team1 came up.

**Now let's see what is Team2 came up.**

Now Team2 recommended to have a separate microservice for the saving account and trading account.

And very similarly, they also recommended separate microservice for cards and loans.

So Team2 came up with four different microservices, whereas Team1 came up with two different

microservices.

**Now let's see what is Team3 came up.**

Now let's go and see what is Team3 came up, Team3 came up with so many microservices like

one for saving account, the second one for trading account, which is very similar to team2.

Now coming to the cards, since the bank application supports both debit card and credit card, they are recommending a separate microservice for debit and credit card.   
And inside loans also, the bank is going to offer many types of loans like home loan, vehicle loan, personal loan for all these type of loans.

Team 3 recommended separate microservices.

So these are the three options that three different teams presented in front of us.

So now let's try to act as a CTO or CEO.

So with the very first option we have drawback because they are not rightly sized.

As you can see, cards and loans, they are clubbed and similarly saving account and trading account,

they are clubbed.

With these, there is a tight coupling between cards and loans and saving accounts and trading account.

And in future when a cards team they want to enhance the microservice for their own needs, then definitely there will be some issues from the loans team.

That's why this right sizing is not a good sizing, at least to me.

I hope you are convinced with that.

Now let's look at the second option.

The second option looks most reasonable and correct sizing to me because all the business domains they are recommending as independent modules or microservice.

So there is a loose coupling and at the same time, this sizing gives a flexibility to the different

teams inside the organization to have their own enhancement life cycle.

And on top of this they can choose their own language, their own database.

Now coming to the third approach, this also makes sense, but in order to go for this service, you

should have huge functionality that separates between debit card and credit card and home loan and vehicle loan and personal loan.

If the functionality is very similar, then this is not a right sizing because you will end up having

too many microservice which will attract lot, many operational overheads.

That's why here my choice is going to be the option two, which looks safer for me right now.

Maybe in future if there are some issues like cards, we cannot put everything inside a single microservice instead, we should go for debit card and credit card microservice separately.

Then obviously in future, after discussing with all the stakeholders, we can separate a single cards

microservice into two microservices like debit card and credit card.

So the overall summary that I'm trying to convey with this exercise is no sizing is a right sizing.

Initially, and if there are some issues you should continuously follow sizing your microservices till

you reach to a safer and correct sizing.

And in fact, the expectation also is not to identify the right sizing on the day one itself.

Companies are organizations they will have their own learnings with their own microservices sizing post that based upon these learnings, they will always try to right size and identify service boundaries for their microservices.

But for now, Team 2 is a winner here.

As a CTO or as a CEO, this makes sense more to me because there is no separate business logic as of

now inside my organization that differentiate between a debit card and credit card, home loan, vehicle loan and personal loan.

There might be some minor differences, but I can handle such minor differences with the help of database columns or any other approaches just for the minute Business logic differences.

We should not separate microservices and go for the approach 3.

I hope this is clear.

**Sizing and identifying boundaries with a ecommerce migration use case**



