**Advanced Computer Networks**
**Instructor Doctor Sameer Kulkarni**
**Department of Computer Science Engineering**
**Indian Institute of Technology, Gandhinagar**
**Lecture 57**
**Serverless Computing - Part 2**

(Refer Slide Time: 00:16)


To understand better, let us try to see how traditional application development used to be and the transitions that have been brought about over the years, and what serverless computing adds to this. Traditionally, application development had been like a monolithic architecture where all the services would be bundled into a single package and run as one process. What it means is if you think of a web application that you would want to run, you typically will define the APIs that you would want.

And if you think of let us say, a traditional store or a shop on which you want to do the e-services, you would also have some sort of a payment that you would integrate, and you would also have the frontend on which the users would interact and have the ability to select, do the search or whatever the options.

And for also the users to log in, authenticate themselves, and then do whatever services they would on a website, and then all the data in terms of whatever the activities that would happen or

the catalog of the information that a system would want to store would all be fetched from a database.

So, in a traditional monolithic architecture, all of these things are stacked in a single vertical integrated unit so that you would run them as one unit together. And what this means is that the way the users would interact, the frontend is tightly coupled with the way you will have your backend, that is, the business logic where it would run and also the database that would typically run.

And all of this evolved over the years, and now we see a lot of 2-tier and 3-tier architectures in web development wherein you categorically sort out the frontend and the backend business logic where the things would run from the database so that you have the very standard 3-tier architecture in which these applications can be built and that is the typical transition we have seen.

(Refer Slide Time: 02:06)


Now, what these monolithic applications really bias is basically to say that it is very simple to develop and deployment perspective because it is a single point of thing where you have to deploy and run. It is very easy to deploy as well. And since all the logic is contained within the same thing, it would be very efficient in terms of how to use the resources as there will not be any additional communication overheads, and all of the resources can be managed in one place.

And likewise, from a developer perspective, it will also be very easy to debug because we are building such a tightly coupled system where it is easier to navigate and see how things are flowing and tracking all in one go. But they bring in a lot of issues, especially, when it comes to handling the failures because now, if one of the units fails, the entire system fails.

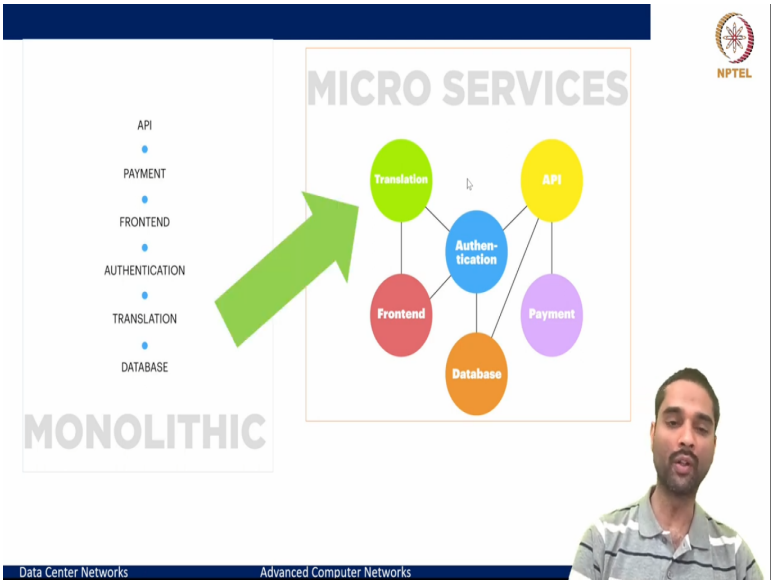
So, there is no way that we can isolate the failures because everything is tightly coupled and integrated as a single business logic, single frontend, all of this in one place and one application. And this also means that it is going to have very poor modularity. That means the teams if they have to work on different things together, it makes it more difficult to again reassess, integrate all of them. Hence, they lack the modularity aspect.

And if we see that your business logic is thin, but the access to the database is a very heavy one and that is the one that is creating issues with respect to latency and performance, you cannot scale out just the database part of it. You will have to scale the entire unit together because it is a single application bundled consisting of all three components.

We cannot scale out just the individual components, and we will have to have the entire application happen for these units. And also because all of these have to be contained, built together, they will also encumber long build times. And hence, these monolithic applications in one way are useful, the other way they have a lot of disadvantages to speak of.

And hence, we evolved from the monolithic to the 3-tier where we can have sort of isolation where the modularity can be generously used, and we are also able to some way scale up just the database entities, leaving the business logic the backend and the frontend the same. So, there have been changes that have enabled us to overcome these aspects.

(Refer Slide Time: 04:28)

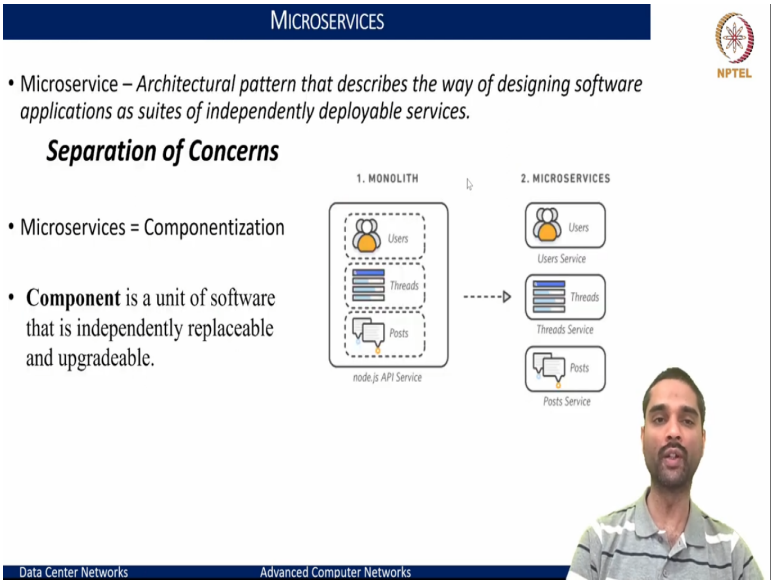



But what is more important and prominent what has taken shape is what we call as microservices wherein you do not want to think of each of these units as a single application but these are all the components of an application that you would want to run.

And if we just consider the frontend side, that would have the authentication as a service that you would want and that would also mean that you would have some sort of APIs that you would run when you want to make certain payments, so a gateway for payment and likewise when you are trying to authenticate, you may have some back-end database through which you are going to access the information and build.

So, a single part of the web application logic can be broken down into a bunch of distinct services, which can then work together to provide a kind of service, and this architectural pattern is what we call as the microservices model.

(Refer Slide Time: 05:23)



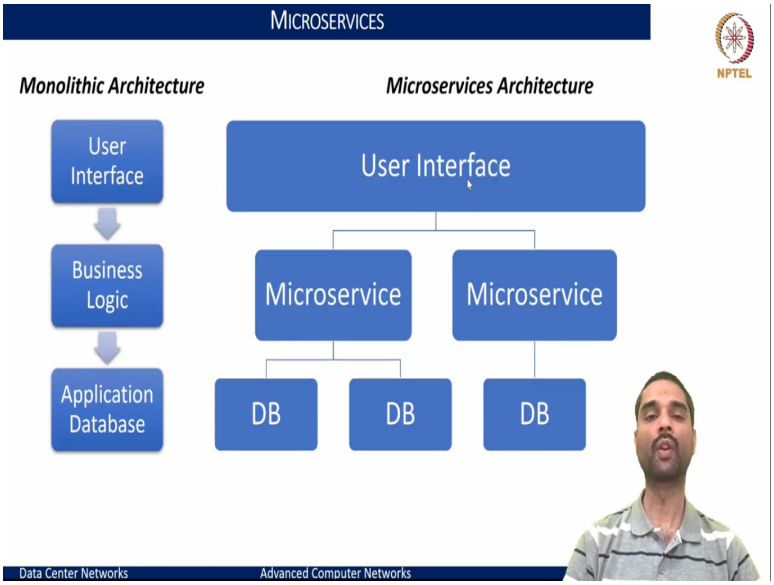

And the core here with respect to the microservices model coming from the monolithic design is to design software such that you have the separation of concerns. What that means is each software application would then have independently deployable services. So, what we are essentially trying to do now on the software is to decouple all the concerns and componentize such concerns into one service and call that as a very microservice.

So, a bunch of microservices would then provide what we see as a single service. So, if we think of the same application logic that we discussed earlier, so, if we think of a monolithic where all of the like the front-end, back-end, the business logic, and all of these associated things are put together, we can decouple them and run as distinct services like users as a user service that can be decoupled where you can scale just the user component which would enable multiple users to connect.

While the underlying thread service, which is going to be facilitating the connections for the users to make some updates or do some additional business logic, can be a separate component that can be scaled independent of the user's component. And likewise, if there are any post services that you would want to add like typically when you tweet or when you go to Facebook and add some note or a message, all of these posts can be handled by a different component in itself.

And that is where the essence of microservices to scale each of these components to work in a distinct function, and to scale these separately helps build more robust microservices.

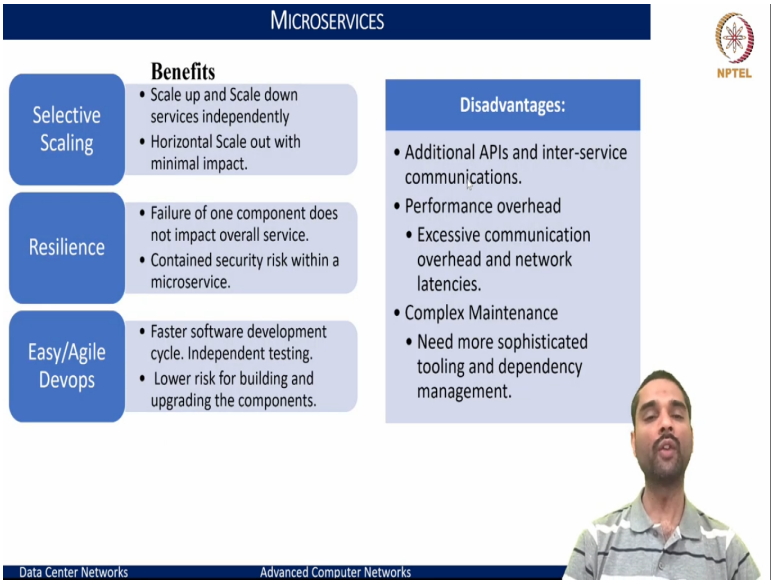
(Refer Slide Time: 06:58)




So, if you think of what we just discussed as microservice versus monolithic architecture, we can build the business logic as a bunch of microservices. You can build the application database as the units that can be independently sharded or built so that you are able to access listing databases for each of these services. So, we can decouple the need for using the same application database for all the services.

So, each service may have its own kind of database that it sees fit. Because sometimes you may want the relational database, sometimes you may want to NoSQL database, and sometimes you would rather have a graph database. So, this decoupling ensures that you are able to match whatever requirements that you need for a specific kind of microservice and bundle them together to facilitate the same service that you would otherwise meet in the monolithic architecture.

(Refer Slide Time: 07:52)




So, the key benefit that we get is selective scaling in terms of being able to scale up or scale down the services independently. And when I mean scale up or scale down, the resources that you would want to allocate for such a service can be added, or that is scale up, when we want to say that we want to increase the memory size or increase the number of CPU cores that are allocated for a particular service, this can be considered as a scale up and scale down.

Second, we could also scale out and scale in the components. By that, what I mean is if I had a microservice for users, I could replicate these n instances of that so that now we are able to scale out these services into multiple of the instances that can work together. And whenever there is no need for multiple of the instances as the users diminish, we can scale back in to minimize the number of services that we would want to run. And that is the major benefit that the microservices offer in terms of facilitating selective scaling.

Second is the fault tolerance and resilience to the failures. So, if one of the components fails, it is not going to impact all the service. And the failure is just contained to that particular component alone. And what that gives us flexibility is we do not have to re-spin and restart the entire business logic, entire application as in the monolithic case. Instead, we would just have to re-spin and start the component that has failed. So, this also minimizes the downtime that you would need to otherwise spin up the right component.

And second, if one of the components gets affected or has some security risk, then it is contained just to that particular component, not the entire application that we can think in the traditional architecture. Hence, it automatically provides this resilience. And also, if we want to change a particular version of a particular service, the upgrades are meant only for that kind of service, not necessarily that because of the change in one service, there has to be a change that has to percolate into other services.

So, this also means it brings in a very faster software development life cycle, independent testing capabilities for the components that we are going to build, and also offers very low risk for building and upgrading of the components. Because we are now focused on just one unit that we can thoroughly make the changes, test it out, and then plug it in.

And we can plug with the other components as it is. And we can also support versions of different components to be interacting and working together. Hence, microservice architecture has been a great success, and this is where containerization along together with microservices have reshaped the way we can think of the recent cloud services.

And nonetheless, these come with certain sort of disadvantages as well. And the major one being the communication overheads that these microservices add. Because now we have decoupled all the components, and they have to exchange information between them. So, that means there is a need for newer APIs for each of these services to interact among themselves.

Not necessarily the standard APIs that you expect to be given out from the users, but these are custom APIs that are going to be built to make each of these service components talk to each other. And if they have to talk, there will also be communication overheads. And if there is a communication overhead.

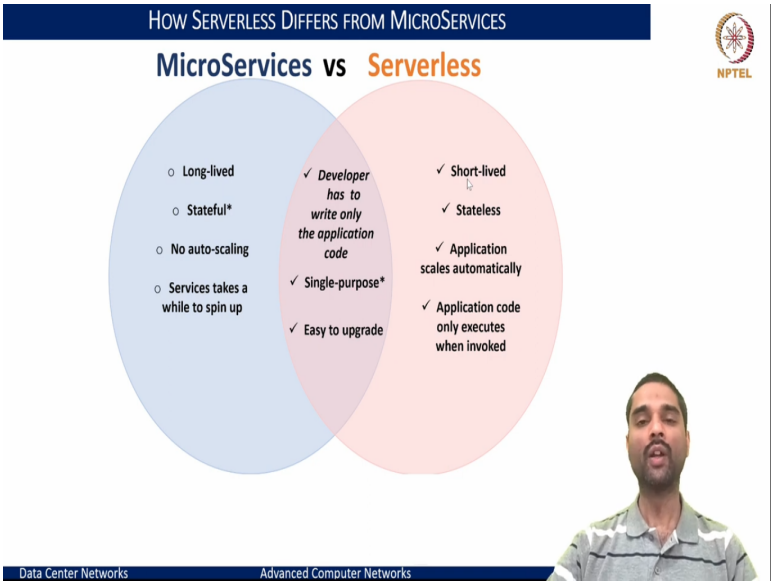
And there is a need for the exchange of data; this will also mean that compared to the traditional architecture, it will result in some sort of performance overheads, in terms of latency as well as in terms of the speed in which the execution efficiency that we can measure. And also, now we can see that if we have different services that may be spread across the network, we will also end up having network service usage or more resources that you would otherwise need and also have these network latencies to add up.

And you are also now bringing up the issues of the failures that can happen for different components independently, and overall the service may fail because of one component that is somewhere on the other edge of the network might have failed. Hence, it also adds to the maintenance problems.

So, the complexity of how to manage and maintain these microservices also becomes a concern. And since we also spoke about that we can scale out these resources, then the maintenance in terms of when to scale out, when to scale in, all of these aspects also have to be defined and managed.

Hence, we would need sophisticated tooling to say how we can monitor the resources, how we can build these management frameworks around them. So, all of these are certain challenges when it comes to the usage of microservices architecture.

(Refer Slide Time: 12:33)




We have seen that microservices have been beneficial, and we want to know like how we can overcome the disadvantages that we are thinking and, what we can, and what are the ways to go forward. So, let us try to see microservices and compare that with a serverless model. So, we saw that microservices enable the developers to write just the application code for a particular kind of service.

And each service is in essence, a single-purpose service that would be built to do just one kind of a job. And also because we are able to focus on just one particular kind of job each time, partitioning each of these services independently, it makes easier to upgrade any one part of the component much more easily than changing the entire application.

And on the counterpart, we see that these services, when we build and deploy, they could run forever, and likewise, they are called the long-lived. And these microservices would also contain some sort of a state while each of the services may maintain its own independent state. And when they maintain their own independent state, it also means that we have to consider the state into effect in terms of as data associated with that microservice when we want to instantiate newer instances.

So, how do we ensure that the state is replicated and made consistent with all the other services that we want to build, and that limits the way we can auto-scale? We can scale only after we have ensured that the state that is being there in the current microservice can be replicated consistently to the other instance, and then it can take up. So, in one way, it allows for scaling, but it does not allow that we can scale it instantaneously. There is a precursor to doing such as scaling.

And also these microservices, they do; when we run them as containers, we would also want to see like they need a certain time to spin up and work. So, if we have these kinds of aspects and what we can work through to ensure that we can or retain the benefits that we get and try to fill the gaps in the microservices so that we get a better model and that is what the serverless tries to do wherein, it retains the key aspects, the beneficial aspects of the microservices, while it tries to patch or try to work around some of the aspects that we see as unfit from the microservices point of view. And in that sense, it retains that aspect of having a single-purposed code and very small specific functions that you want to build.

And instead of making them long-lived, the principle of serverless is to have short-lived functions. That means the service should only be working when necessary. And this short-lived concept also means that we are going to save a lot on the resource consumption that we would otherwise have.

And when it comes to the user's point of view, you would not have to pay for the resources when you are not using them. And it also provides flexibility to match and basically ensures that we are able to consolidate and run multiple of these services when possible. Because each would run and take just the amount of resources they would need.

Otherwise, if you are running, keeping a container continuously running, you are in some way eating out the memory and not allowing other services to run when the memory could have been given out to some other services. And that is where the short-lived concept of serverless really helps.

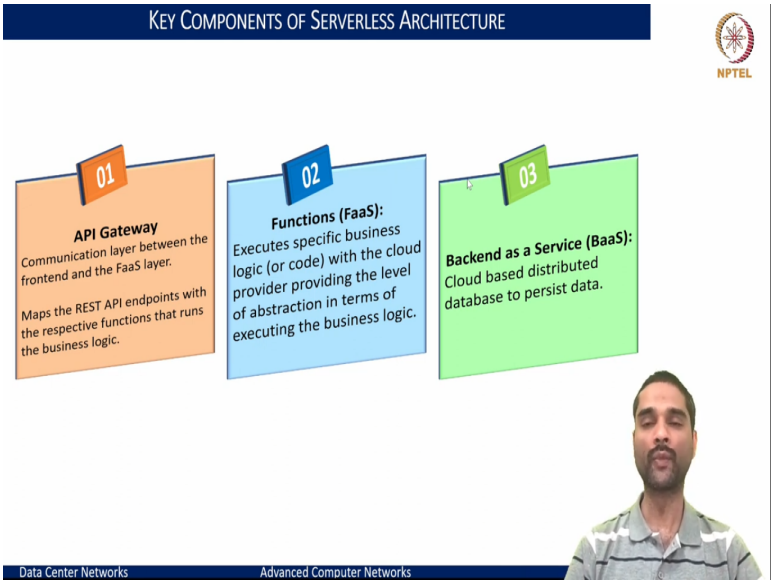
And second, because we were stateful, we had to ensure that the state is coherently updated and made consistent before any new instance could be taken up. The serverless comes with a philosophy of making these microservices to be stateless. And this statelessness is a very important characteristic to say that now every instance can come up and be run whenever they need, and go away whenever we do not need them.

And there is no need for any additional overheads that you would need to synchronize the state. So, in essence, if I take out the state from the microservices and run them whenever I just need them, I make these microservices as a serverless model. And such a model would automatically allow us to scale without worrying about any of the other aspects of state synchronization. And this means now we can apply auto-scaling on these functions.

And since the auto-scaling can be done, we now can have a logic to say how to scale up, how to scale out, how to scale in for the specific resources, for the specific services which we so now call as the functions to operate. And the application code would only execute when it is invoked. And this is what ties up with making them to be very short-lived.

And once they are done with the execution, they would go away. And all of these are the key principles of how the serverless which tries to build on the microservices framework to add these potential benefits.

(Refer Slide Time: 17:37)




And to build such a serverless framework, we need to understand then what are the key components. The first is the API Gateway. Here, what is, in essence, necessary is to say that I want to invoke a function only when necessary. That means somebody has to trigger the call to this community function to start it.

And that is done basically by means of having one entity, which we call as a gateway, to be the always-on component which acts now as a communication layer between the frontend and the function service. And this communication layer is the always-on component to which any of the requests can come.

So, I can have n number of microservices which would all be behind this API Gateway which would be invoked by this API Gateway as and when it receives the request. And this communication paradigm in terms of when, how it receives the request is again based on the rest API aspect of how the HTTP transactions take over the web. That is, in essence, the rest API that he will communicate with the server to access particular services.

In this case, the server is the API Gateway to which you will request access to particular functions or the particular services that you would want to run on the business logic. So, the API Gateway is, in a sense, the gatekeeper which would run the necessary custom functions within the system. And the next component is the functions themselves, and these are the functions that

a developer would develop as microservices, which would be run and executed only when invoked from the API Gateway.

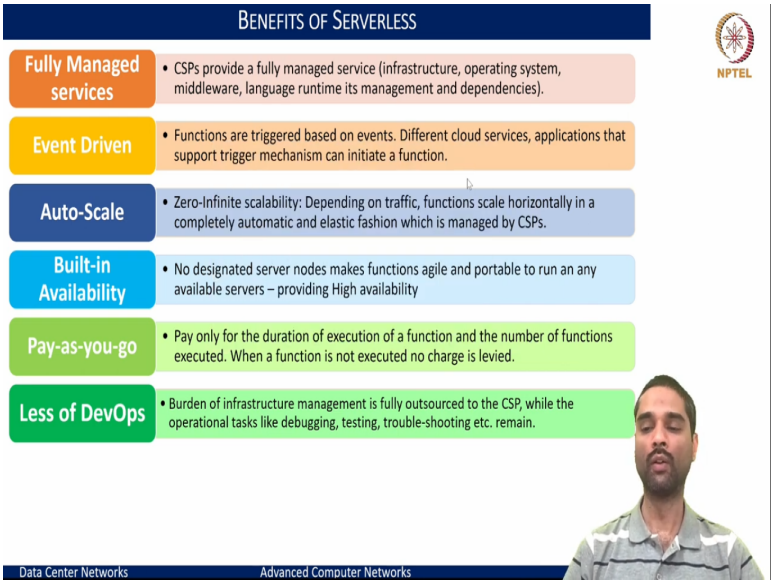
And these functions would typically constitute of the business logic or the code which were to be run on the cloud service. And in essence, this is creating the abstraction of how the business logic would be executed. So, you are saying, I want to invoke this function and pass the necessary arguments: the data, and then it would operate and then update back the information in the rest API model.

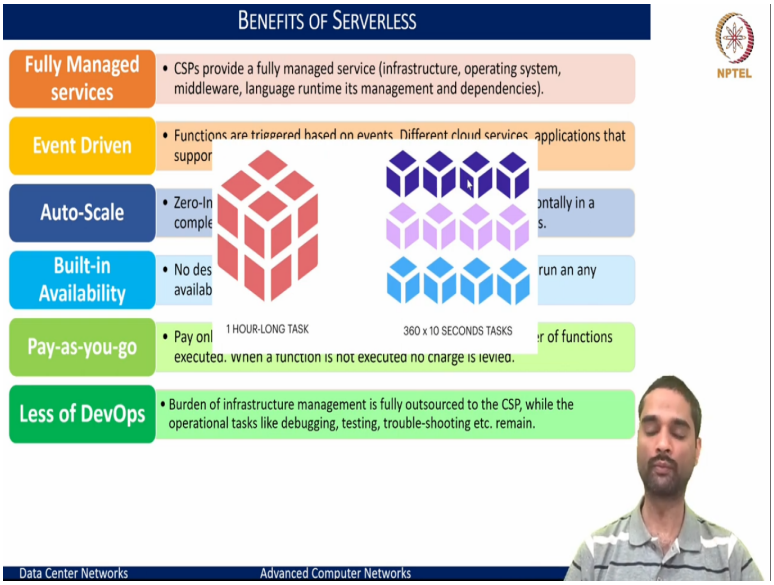
So, you have a request, a request would be executed as a function, the function would execute and send a response back to the API Gateway, and the API Gateway would then communicate back to the user. And we said these functions are stateless. But in reality, whenever we interact with any of the web applications, we know that there is some sort of a state that is maintained between the client and the server, like the cookies that we speak of where the entire information of state is being pushed onto the client's browser.

But nonetheless, the server also updates and keeps its state in its business logic. So, we can still retain the same information the way how the cookies on the client side would be used to maintain the state. But what about the state on the server side? And to do this, we would need some sort of a backend as a service or where you have some sort of a database on which these functions, no matter how many instances of functions we would run, would all go and update the state onto this backend service.

And this is called as a backend as a service or the BaaS, where you are basically going to have it as a distributed database where the data would persist. And irrespective of the function life cycle, this data would continue to persist and be made accessible to any of the functions. And these in essence, form the means for us to build a serverless computing architecture.

(Refer Slide Time: 21:08)




And through this, we would also want to understand what kind of benefits we get. The foremost is the fully managed services. What this means is the cloud service providers now have the ability to provide a fully managed service on top of their infrastructure, operating system, and middleware, including the language on which the users would want to write and develop their code.

So, we can say this, in a sense, similar to what software as a service provides. But unlike where the entire software is also going to be governed and dictated by the cloud service provider, here,

the flexibility is given for users and developers to develop their own software and deploy that as a service on top of this framework.

And that is where it makes it more essential to basically think of I write a code and then deploy it on the system to run. And the entire management of such a system would then be taken care by the cloud service provider, while the user only takes care of writing the logic for the code.

Second, this model is basically what we call as an event-driven system wherein the functions, even when I deploy, do not mean that it is up and running all the time. It is going to be triggered or invoked only when there is an event that necessitates that function to be run. And different cloud services and applications that essentially require the trigger for such an event to be generated have to be built around when we want to have these serverless functions supported. And we will see like in Amazon, in the Google, cloud functions Azure, all the frameworks try to build this event-driven system in trying to facilitate the user to write the serverless functions.

Third is the auto scale model, in terms of having, or we can say, we can scale out to an infinite number of the functions that we can run, and we can scale into as low as zero functions that would be on at any given time. That means it gives the full width of 0 to infinite scalability. And this can depend on the traffic that we would see at any given time, and we can match to any sporadic traffic requests or certain bursts of traffic that would come. And this horizontal and elastic scaling would then be managed completely even by the cloud service providers.

And the user would only have a choice to say at the reasons of how to set up the scaling, what is the limit if any because of the cost constraints that you want to consider in building such an auto scale framework.

Next is also the availability aspect. Because now a failure of one function does not mean that service has gone down. I can respawn the function again or create a function instances as and how many we want. We can even run replicated functions that would run and while one would work, the other can also work in redundant model and build the same services. So overall, this ensures that we are going to get better availability and fault tolerance aspects with the serverless.

And as we said from the serverless, they are invoked on an event and would run for the time that they would need and then go away. That means the usage of such serverless functions is confined to the time that they are going to be executed. So, the payment perspective you would also consume the resources at a limited time. And you can pay for only the duration of the execution of a function rather than having to pay for the VMs that you would otherwise keep running all the time or the containers that would just keep running all the time without having to do any active work. And this is again a major incentive in terms of how it cuts down on the cost.

Nonetheless, this was debated heavily whether it is really helping or not. Because different cloud service providers would use this as a choice to say this is a better effect. But in terms of the charge, maybe they are charging you more, maybe the benefits that a user may see for running large or long instances may not be really adequate to say that it is beneficial.

But this was something as research many of the papers were dawn and see which is better, whether platform as a service or serverless, what are the cost overheads constraints. But all of these, in essence, point to maybe there is a need to how we build the economic model around the aspect of serverless functions.

But the whole concept of pay-as-you-go is a very clear and nice means to say that you would need to pay only when you are accessing these particular resources or services in a given cloud rather than paying for just renting out certain things in the cloud. And also when it comes to the management from a developer's point of view in terms of the continuous life cycle of software development, you would want to have the means to build the infrastructure where you are able to have better devOps.

And that means less of the overheads in doing the devOps for managing these cloud services. And serverless helps a lot in terms of alleviating the infrastructure management first. Because now you are only concerned about the code that you would want to build. And the entire infrastructure management is now going to be taken care by the cloud service provider.

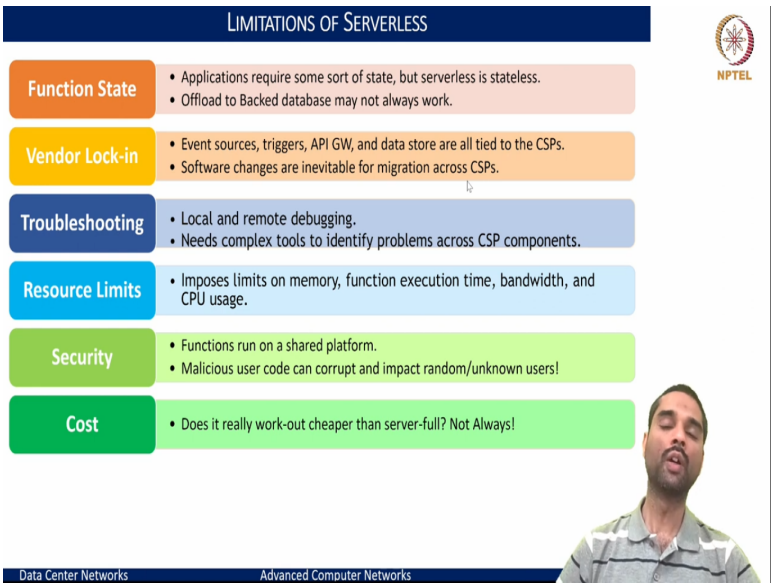
So, the better energy focus can be on like how to build the better debugging, testing of just the code component that we will be building. Hence, this aids directly to the less of devOps. So, to

give a perspective, let us think of like you have a long task that you want to run on a cloud. Let us say one hour-long task.

But in essence, it is basically an accumulation of very small tasks that you could do. And if I had a monolith, you would need one hour to build such a task and execute it to completion. Now, if I had built them as microservices of independent aspects that can run and if I can further build those as independently scalable tasks, then if each task takes just around 10 seconds, one hour long task would mean that you have 360 of such tasks to complete.

And with functions, if I am able to scale out to 360 functions while my resources support, then I can execute the same in just about 10 seconds. And that is the benefit that you get as long as there are no constraints with respect to how the scale-out can happen. We can scale out to as much as possible to minimize on the latency to provide better SLOs, and meet the client's QoE requirements. All of these can be achieved by a serverless model.

(Refer Slide Time: 27:43)




Nonetheless, they have certain limitations. One like as we said the functions cannot have any state associated with them. And if at all there is a need for state, they have to be offloaded to the backend database. And what that means? If your function is going to be repeatedly accessing certain kind of a state and having multiple states to deal with, you will have lot more

communication overheads in terms of communicating with the backend database, exchanging the state information back and forth.

Second, is about the vendor lock-in that you may end up having. Because we said when you build a serverless function, it requires certain runtime support that you would want. It requires the API Gateway which is going to trigger and run this function. And you would also need the backend database that you would want to communicate and work.

So, how these event sources can be plugged for a function? What is the trigger point through which you are going to come up? All of these are in some way customized and custom for typical cloud service providers, like Amazon Lambda, you may have different event trigger mechanisms, in the Google, you may end up having different event triggers, and likewise in Microsoft Azure and other cloud service providers.

And what this vendor lock-in in such aspect means? If I deployed my code on Amazon Lambda and it is using SNS or other kind of a service, now, when I try to migrate to the Google or Microsoft Azure, I have to re-adapt the SNS service to the corresponding service that a Google or Microsoft offers, in terms of, it could be the email service, it could be notification services, any of them. So, in essence, the decoupling of the function with respect to the entire set of services, but now, the coupling with respect to the vendor-specific APIs. This is where the challenges may start to creep in.

And third is the troubleshooting issues. Because what we see is, we have lot of the functions that you can deploy but when the things have to run, if there is a concern somewhere in the API Gateway, the function is completely going to remain agnostic to what is going on in the API Gateway. And if there is some bug in the API Gateway in terms of how the data is going to be transformed and provided to the function, it may become very difficult to debug and say where the problems lie. And likewise, when the databases are going to be updated or any custom vendor-specific services are going to be used, all of these are, in essence, independent components, and this integration of several of these components can also create barriers or troubles in terms of trying to troubleshoot where the things are going wrong.

Because things in isolation might be working fine, but things and integration may not. And that is where also this is again another technical as well as research challenge to say how do we build the tools to dip that can help better troubleshoot these serverless functions.

And next is about the resource limits that we spoke of like we said we can scale infinitely. But in real world, it is not going to be possible at all. You always have the limits in terms of how much memory you have in your system, how many CPU cores that you have, what is the disk storage that you have, and how many database connections that you can make with respect to backend. So, all of these constraints are somewhere now hidden from the user point of view.

So, from the user point of view, you are building just the function, you can see how much time it takes for the function, what is the communication bandwidth requirements that it may have or CPU usage requirements that it may have or the memory requirements that it may have, but not have any specific control, in terms of, how to dictate these parameters themselves. Hence, the resource limits could be another major concern.

And also, when we speak of the cloud service providers where we are trying to run these functions, they are going to be run on a shared platform. So, how do we ensure now for the security and privacy aspects of these functions? What is the guarantee that your function would not be run for someone else's services, and someone else's service may even try to corrupt your function. Like when it is invoked from some malicious user maybe it would create some wrong side effects. So, how do we build the security framework around these serverless functions is also an important aspect to look at.

And like I mentioned earlier also about the cost, it is always a debatable aspect in terms of how different cloud service providers are charging and for what aspects. So, at times it may work out better but at times it may not. So, what kind of a services really fit for the serverless, where it would not be an ideal fit, needs to be understood by the user before moving for these serverless applications.

(Refer Slide Time: 32:31)


So, let us try to see when we should be going to serverless and when we should not be considering serverless as a model for deployment of the cloud services. And the good part of the services that would fit in the serverless model would be those where we see those as a short-running, stateless, and event-driven services.

So, think of when you are on a tour, you take a photo, and this photo needs to be updated on your cloud drive. So, this is kind of an event that happens where you take a photo, and then you automatically want to run the service that it would update. So, you do not need a VM that would be running all the time in the system.

But you want just the photo that you take at a given time and then update it and keep it. And even when you want to store, maybe you want to do some compression, maybe you want to do some quality enhancements, different kinds of functions that you would want to run on the photo service.

So, all of these can essentially be like basically when you take a photo, it triggers an event, and you can run all of these independent functions and then store the data on the cloud. So, such an application would really fit as a best fit for the serverless model. So, to consider in generality like any kind of a microservice or any of the apps that have the mobile backends that you would want to build and ML inference services.

So, you are looking for certain things, and you want to find out what kind of image that you captured, what kind of data that you have. So, ML inferencing where it runs on an instance basis, which can scale as you need or you want to have the bots that would have the chatbots that you would want to build in terms of how to communicate with the users, all of these which are essentially event-driven.

And when we think of IOT, most of the use cases are event-driven, wherein certain activities happen, the actuator triggers, and then you have certain processing to do, which are basically short-lived functions that you would want to execute in the backend. And this is where the IOT would really also be a nice fit for serverless functions.

And stream processing to some degree, like very short clips that you would want to run which are going to be run like a 30 second or small clips or like the tweets that you have, all of these are kind of very small scaled, short string processing could also very better fit for the serverless.

On the other end, if you see the very exact opposite, where we have very long-running services which essentially keep the state information over a long period of time and also require the services to be latency critical, then they would be the bad fit for the serverless. So, to think of, if we have any database as a service that you would want to access, keep the data; you typically keep a long-running connection with the data and keep updating the information.

And if I think now as a database, where I want to provide as a service, making this serverless would mean a lot of overheads that you are trying to bid. And especially in terms of making that you have to connect, disconnect which is a major challenge, and you typically want the databases to also be cached so that you are able to access them back and forth again and again whatever the data, but if these are going to be accessed every time afresh, that means you are going to pay the penalty for such access to the databases.

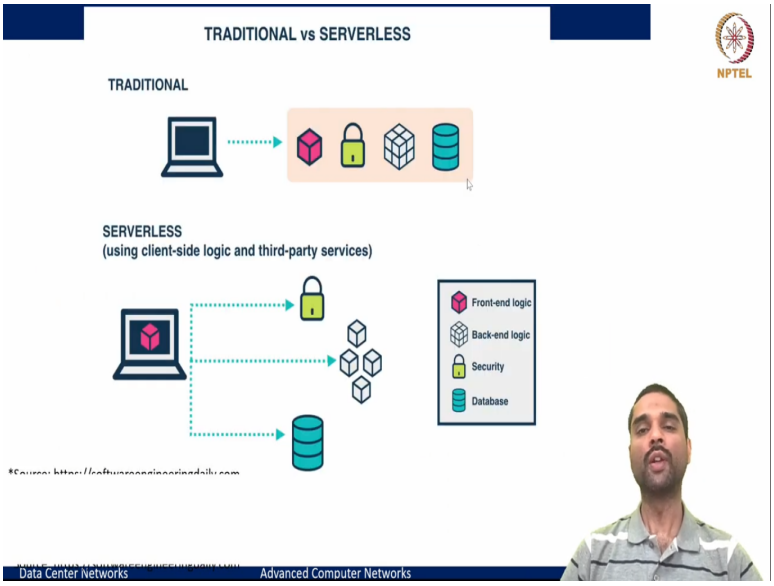
Hence, these would not be the right fit. And we spoke earlier about inferencing, which is going to be done on an event basis. But when we think of training, there is a lot of information that goes back and forth in terms of what training parameters you are updating, and there is this back propagation that also happens.

So, there is a lot of communication involved in the training process, and these are typically run for hours together and days together for having these deep DNNs or deep learning models to be trained over time and hence, these would be also a bad fit if we think of these as applications for serverless.

And heavy-duty stream analytics, if you have an engine that is continuously going to operate on a particular kind of data and do some data analytics, this such a stream analytics platforms like Spark or the Storm or Apache Kafka, all of these stream processing platforms would not be the right fit in this case.

And any of the numerical simulations that we think of that are going to take a lot of computation time, in terms of being compute heavy and would run forever to do certain jobs then running those as the serverless functions would not be the right fit. And likewise, the long video streams that you would want to run may also not be the right fit when it comes to the serverless functions.

(Refer Slide Time: 37:13)




So, let us try to look at some of the very specific use cases where serverless would be very handy. So, if we think of traditional web applications, that we think, the web applications in essence, your user is connected to those sites for some specific short duration of time, trying to access certain data, do some operations, and then get back.

In the traditional one, you have the front-end logic, backend logic, and security database, all of them viewed together, and the server to which you would connect would then provide you this frontend for the user to work with, and then it is going to provide the authentication service, the access control service which we can club as a security measure.

And then have the backend through which the business logic would be run and your information in terms of if I am you visiting Amazon site then I am looking at a catalog of the data that I want to purchase, procure. So, they would have such a database through which they can provide. And I could have my own cart where I would add the instances and then go.

Now, this if I have to realize in the serverless model, we can try to use the client-side logic and build the third-party services around it. In essence, what we are trying to build is to have the front-end logic that can be built essentially in a different way and have a different service for providing security. And have the backend logic, like if when we think of Amazon as a means to do some event content that we want to buy.

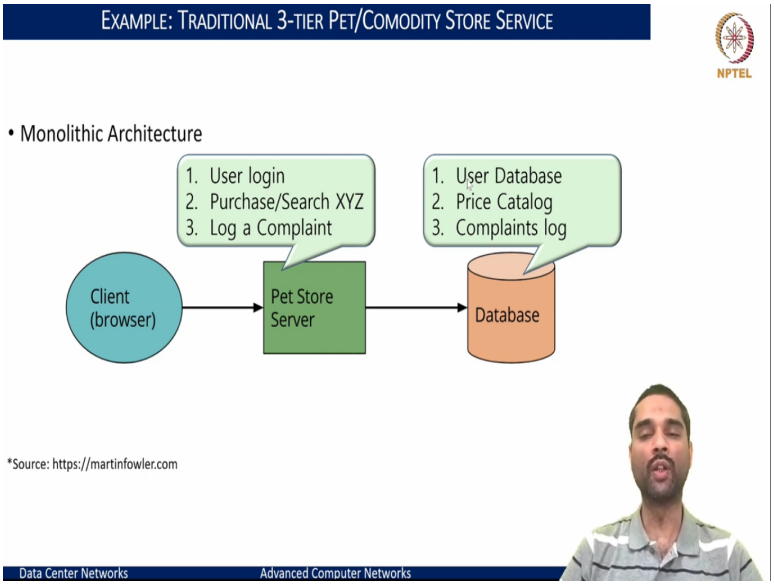
You may have Amazon maintain different categories of data. Like it could be electronics, it could be household items, and all of these can be written as different microservices stencils. So, only when you interact with, you want to switch to the house, and you want to bring that info and build and on the frontend and the way you would want to have a common backend business logic where you maintain the cart, which is another service that would run specifically for a user and have the information persist in a database.

So, likewise, now you can think of a single web application or a web application logic being split into multiple of the microservices that are now going to offer, and they are going to run only when there is a need. So, if I am now browsing for electronics and want to buy gadgets that I am looking for in that category, I can have subsets of microservices that I can run.

Let us say I am interested in buying some tablet, and then only those kinds of products for which I am interested could be delivered as a key service that I could use. And now, if I switch to household items and look for an air conditioner or whatever the other kind of device then those kinds of inventories can again be plugged into separate services.

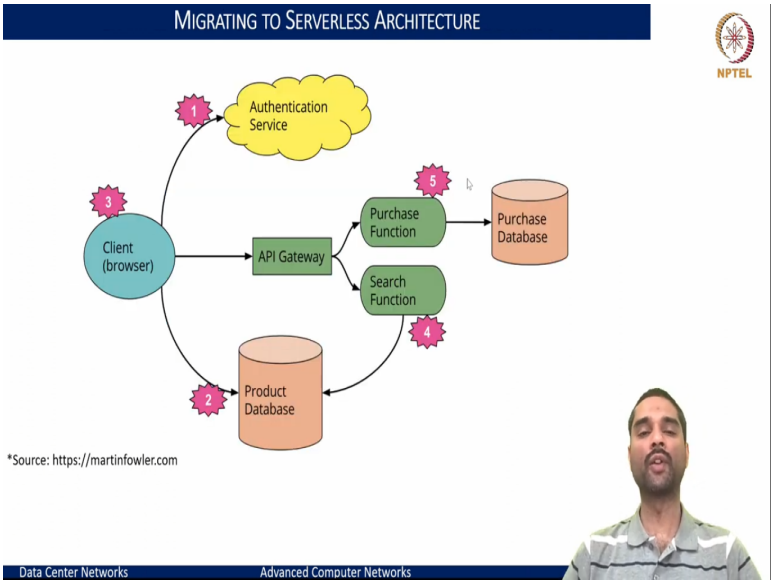
And you now see that any updates that can happen on one end of the list would not impact anything to do with the other end. And likewise, there is a lot of granularity, scalability that this kind of model would build. And that is where we can think of this to be applicable to a typical web service.

(Refer Slide Time: 40:03)




And if you think of any commodity stores, we would have the cloud client browser login, connect and search for the components, and then update, or if you say any issues, you can post the complaints and likewise, and all of this information would eventually go to the database. And a typical monolithic architecture, all of these would be wondered as one series where any of the activities would put use the entire application spin on the backend and the database.

(Refer Slide Time: 40:30)




But when it comes to the serverless architecture, now we are talking of all these microservices which would be run as independent functions only when you would need. So, what would change from the client side is now, instead of connecting to a server, the client is connecting to what we call as the API Gateway.

So, that way, the client's connection would be with respect to API Gateway as if it is connecting to the same website. And then the API Gateway would then invoke the necessary kind of function. So, if a user is trying to do some kind of purchase, the purchase function can be called; if he is trying to do some search for some kind of data on the web page, then a search function could be invoked.

And if there is a need for any other kind of data, like when he wants to log in or leave a comment or whatever, then that can be another function that you would want to run. And all of these now are basically distinct functions that can be run to facilitate the same service. And again we need the database in terms of we can isolate the purchase database with respect to what we are trying to pull in and put the cart for the user.

And a product database in terms of what catalog of information users want to search and they can all be independently managed. Thus, we will see that it gives a lot more flexibility in how things can be deployed in a serverless model.

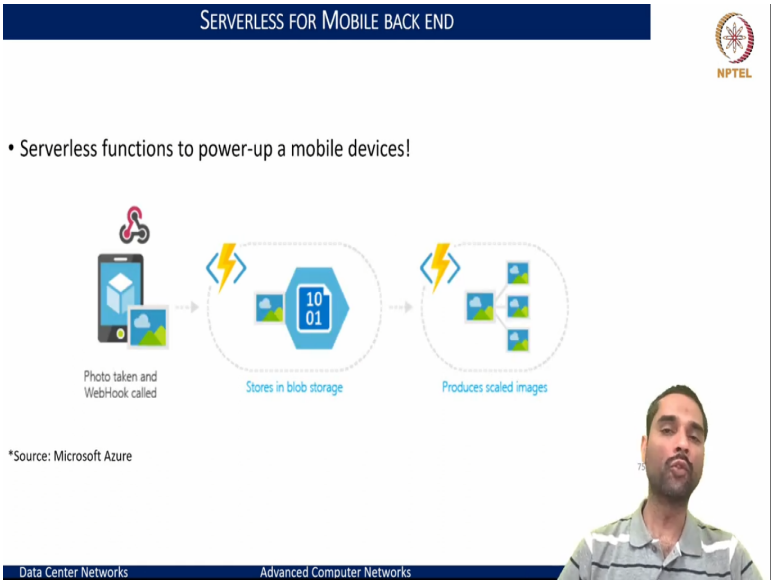
(Refer Slide Time: 41:55)


And any of the web applications, for that matter, would readily fit. And with the advent of the serverless model, like we would have heard, many of the websites moved to the serverless model. In fact, BBC was the one that started this in the early 2015s, and then they had a very successful migration where many of their web page loads which would happen during a particular period of time where it speaks out, and they were not able to scale their resources.

Now, they can spawn in distinct functions and manage the resources, while the rest of the time, the servers would not be utilized. So, they do not have to over-provision the resources either and allow the functions to be scaled as much as the need in a given point in time. And this also creates a lot of flexibility in trying to say how you want to do server-side rendering in terms of when we have the function that would call another set of sub-functions to create and see how the user's page would be and then pass the server-side rendered page back to the user.

And several of the other websites have also tried to move their web applications into the serverless model.

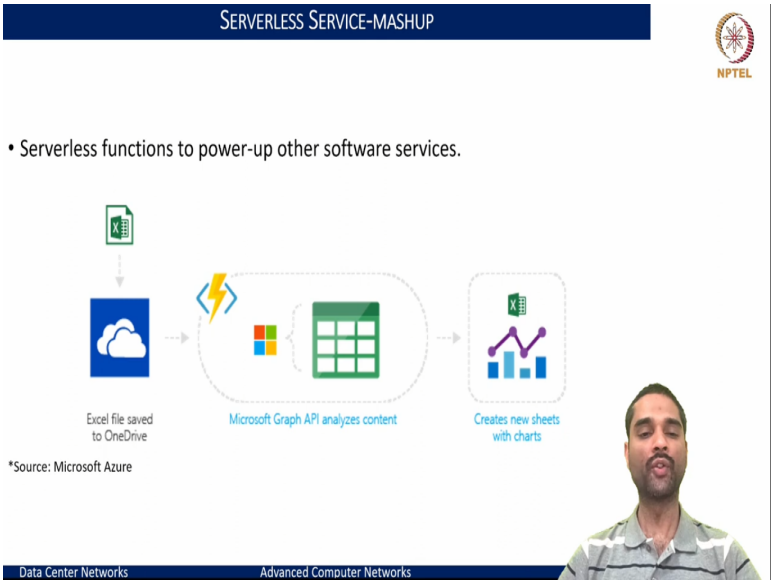
(Refer Slide Time: 43:01)




And I also mentioned earlier about the mobile as backend, for serverless for the mobile backend where you want to take some pictures, want to upload them any of the cloud server repository or some sort of a bucket, and then want to do some kind of functionalities that you want to run on the audio, video or the image data that you would have taken. So, all of these functionalities can be very easily, readily achieved.

And these functions need not be run when it is not necessary. So, you are only doing a pay-as-you-go. And this essentially also offloads your function mobiles from doing any functionality. And in fact, when you take a picture on your mobile, the service provider himself can have these functions, which can be deployed on a cloud to run for you and provide that as value-added services to the user. So, it opens up again a new model of how the services can be integrated onto the mobile devices.

(Refer Slide Time: 43:59)

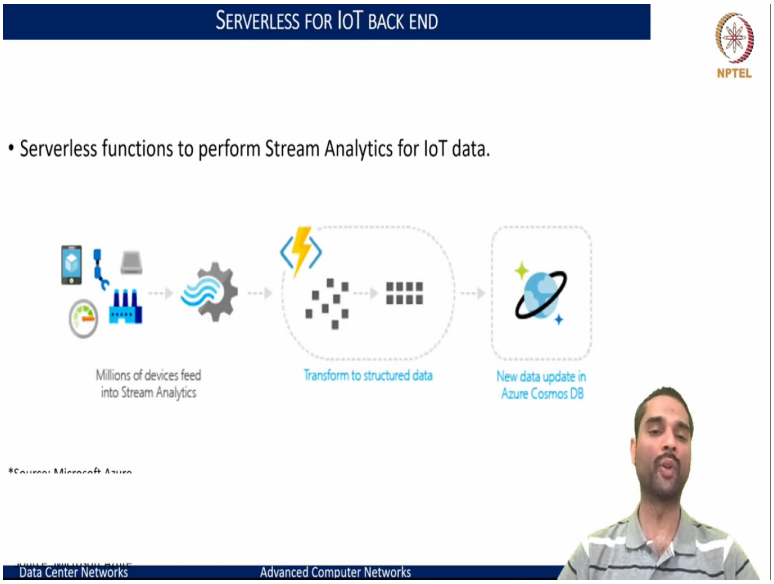



And what this also allows, like if you are dealing to do with a lot of data analytics and you are trying to do a lot of ways or means to represent particular data. If I upload any Google sheet onto OneDrive and I want specific functions to be run, then I can essentially trigger for treating this data and try to do some customizations.

Like if you are having a Power BI dashboard where you would want to showcase the different means of what data it is in different kinds of graphs, we can have the functions to plot the necessary graphs for these Excel sheets, and whenever you upload such a sheet, these functions should be run, and it can create new sheets with the necessary charts.

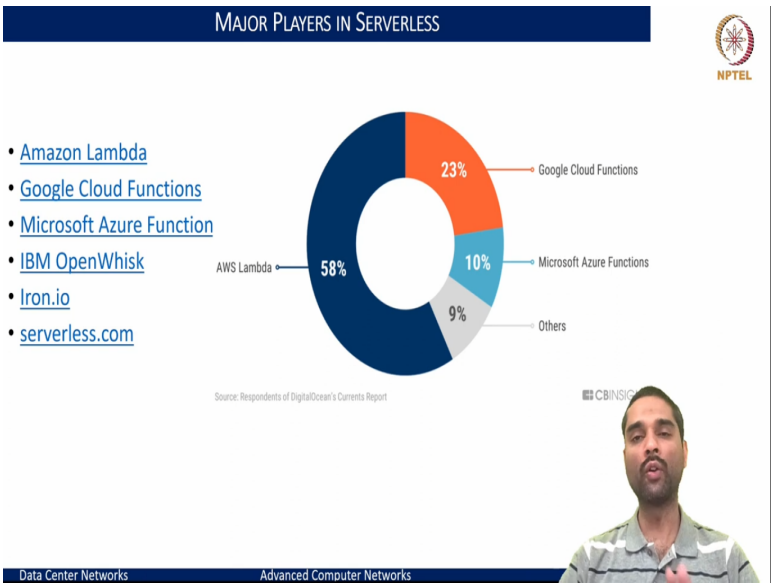
So, any kind of a service where we see that there is a need to run specific functionality when a particular event occurs, and these are like short-lived functions that you would want to run, serverless fits very well.

(Refer Slide Time: 44:55)




And I also mentioned about the IOT backend where, like, if you have some analytics that you want to run where any of the IOT whenever they trigger for a particular kind of an event, you would want to run those functions on behalf of those we collect, accumulate the data do certain processing and then update all of these functionalities can be readily achieved through this serverless model.

(Refer Slide Time: 45:20)




So overall, now, if we look at how these use cases and what are the key players in the serverless world, we see that Amazon, which has started its work as Amazon Lambda or AWS Lambda, the

pioneers have a major share. And then the Google Cloud Functions and Microsoft Azure, and then there are several other players, including IBM, Iron.io, serverless.com, who have also tried to build these services.

And over the years, these numbers have been changing. But we can see that there has been already a lot of push in terms of how these serverless players, whom we can think of as the major cloud service providers, have all started to offer these serverless functions.