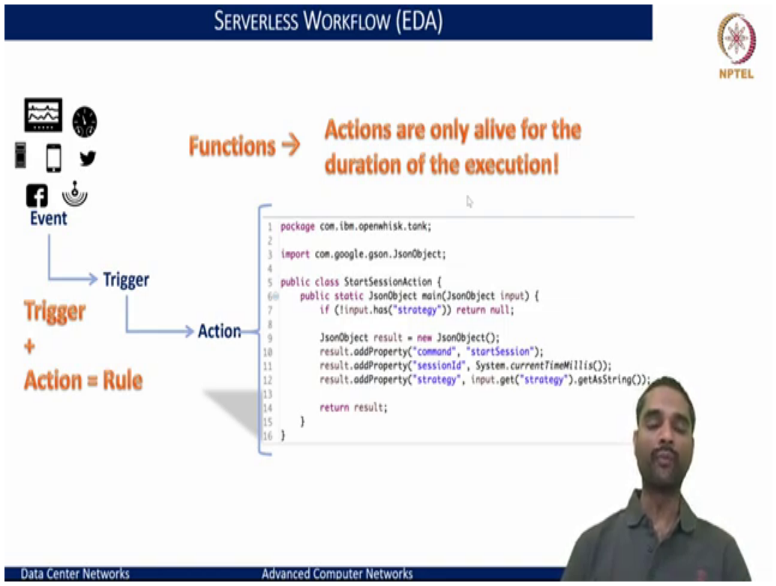
Advanced Computer Networks
Instructor Doctor Sameer Kulkarni
Department of Computer Science Engineering
Indian Institute of Technology, Gandhinagar
Lecture 58
SmartNICs and In-band Network Telemetry, Future of Network Softwarization, SDN 3.0

(Refer Slide Time: 0:18)



SERVERLESS WORKFLOW (EDA)
NPTEL
Functions →
Actions are only alive for the duration of the execution!
Event
Trigger
Action
Trigger + Action = Rule
Data Center Networks
Advanced Computer Networks

```java
package com.ibm.openwhisk.tank;

import com.google.gson.JsonObject;

public class StartSessionAction {
    public static JsonObject main(JsonObject input) {
        if (!input.has("strategy")) return null;

        JsonObject result = new JsonObject();
        result.addProperty("command", "startSession");
        result.addProperty("sessionId", System.currentTimeMillis());
        result.addProperty("strategy", input.get("strategy").getAsString());

        return result;
    }
}
```

Let us try to get some insights on the serverless architecture and what are the fundamental research challenges that lurk in serverless. Fundamentally, this serverless architecture is based on what we call as the EDA or the event driven architecture. What it means is a kind of a software architecture pattern that orchestrates around the behavior, the way the events are produced, the way the events are detected, the way the events are consumed, the way you react to certain events, so everything is woven around the events.

So, in the serverless architecture, what we really see is the source is the event. And once those events are triggered you take several actions. And these actions are essentially the functions that are being built and executed as the model of unit of execution to do certain tasks. And these actions only live for the duration of the execution. Otherwise, they do not have any other means to take the compute resources or occupy memory or any other kinds of compute resources. Thus, what it essentially means is a single event, once it is generated, it can trigger one or more actions. And each of these actions result in the execution of functions.

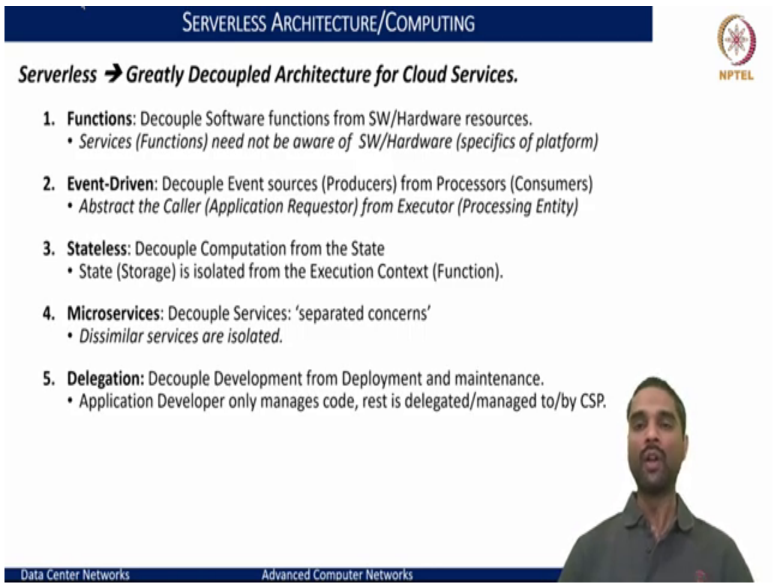
To look at this pattern, what it means is you are observing for the events and that is where EDA kind of an extends the observer design pattern, and there can be multiple recipients of these events, where each event may trigger a different action on different observers. So, as these events are being pushed onto these observers, the observers take necessary actions by executing certain functions.

And this model in a way is you can see that it is asynchronous because the events can occur any time and the actions can be taken as and when these events occur. And it is not necessary that these functions be executed as and when the event occurs, the events, when they are triggered, they may be deferred to take certain action.

That means even the function execution can be asynchronous with respect to events, like when we want to store a movie and then want to transcribe and generate either subtitles or text and then we want to translate to some other language, these functions can be processed as and how the resources become available, and be executed at any other point later not necessarily when you push a new content onto the cloud.

And this pattern allows flexibility for us now to define the functions which could either be run synchronously or asynchronously. And if we are trying to do anything in real time we would want and provide a response back then we would want them to be run in synchronous fashion. Otherwise, we could just store the kinds of activities or events and then pile up the set of events to work on a batch of these events and execute these batch of things as single function. So, both models can be applied with serverless pattern.

(Refer Slide Time: 3:34)



Serverless Architecture/Computing

Serverless → Greatly Decoupled Architecture for Cloud Services.

1. **Functions**: Decouple Software functions from SW/Hardware resources.
   - *Services (Functions) need not be aware of SW/Hardware (specifics of platform)*
2. **Event-Driven**: Decouple Event sources (Producers) from Processors (Consumers)
   - *Abstract the Caller (Application Requestor) from Executor (Processing Entity)*
3. **Stateless**: Decouple Computation from the State
   - State (Storage) is isolated from the Execution Context (Function).
4. **Microservices**: Decouple Services: 'separated concerns'
   - *Dissimilar services are isolated.*
5. **Delegation:** Decouple Development from Deployment and maintenance.
   - Application Developer only manages code, rest is delegated/managed to/by CSP.

To understand more deeply about what this serverless provides, I often mentioned that the serverless is all about having a greatly decoupled architecture for cloud services, and decoupling in serverless comes in various forms.

First is the functions themselves. What do the function decouple, essentially, is the software from the hardware. And we have seen this concept with NFV earlier. And here because a developer is now more focused just on the function or the software part, he can be completely agnostic to the hardware or the specifics of the platform and build the functions the way you would want with respect to keeping the runtime requirements, the API that you would want to use.

Second, as I mentioned this is an event-driven architecture. What it essentially does is to decouple the event source from the event consumer because it could be anybody that could generate the events, which is essentially called the producer of the events. And then once these events are produced, the consumers are those who process these events and take certain actions; these are other ends of the entity. So, there need not be any kind of a coupling to build them together. And the way to facilitate this is to have the rest APIs to communicate the events and then the events can be consumed and do certain actions through the use of the functions.

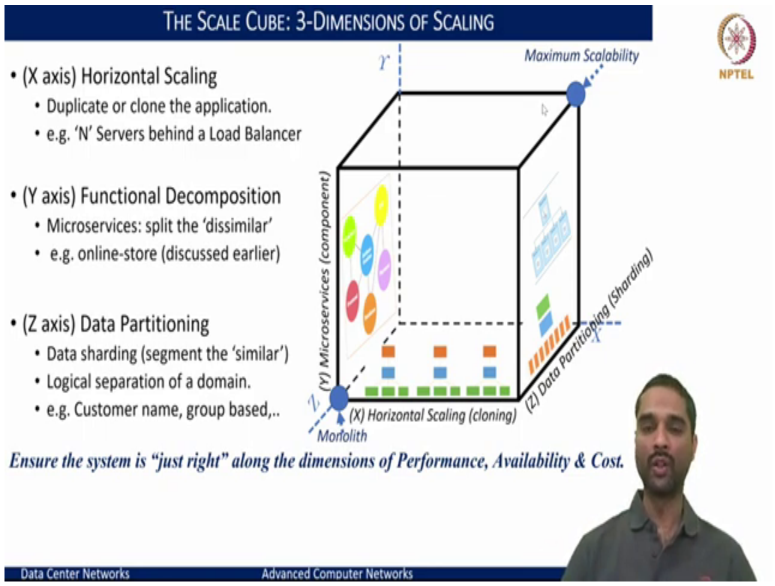
Third, the functions as we said is stateless, we have, in essence, decoupled the computation from the state, because when you execute a function you are basically doing certain form of a computation. And now this function being stateless, that means, we are essentially borrowing state and then updating and pushing it back making the function to remain in its entirety to be stateless. And we are trying to pare it in this way storage or a state that is completely isolated from the execution context.

Fourth, we are using the microservices. And we said microservice is all about trying to decouple the services with respect to their concerns. In essence, we are disaggregating different services based on what aspects they need to really address. And having such a dissimilar services being isolated means we are trying to have much more loosely coupled system.

Next is the delegation. And this is a part of a hidden kind of a decoupling when we see but this is primarily applicable when we look at the monolithic or the traditional architecture with respect to what serverless provides. With the traditional architecture, typically, it is not just the application that you develop you also think about how to deploy how to manage all of these, the nitty gritties come along with the development. But with serverless, it completely delegates the job of management, the job of how you would want to instantiate a function, how would you want to make sure that the function scale, all of these are being decoupled from the development process.

Hence, this decoupling from the development and deployment lifecycle and maintenance cycles greatly enables the speed with which the developments can be done and deployed independently. Hence, we can think of the serverless architecture as a greatly decoupled architecture for the cloud services.

(Refer Slide Time: 7:02)



THE SCALE CUBE: 3-DIMENSIONS OF SCALING

- (X axis) Horizontal Scaling
  - Duplicate or clone the application.
  - e.g. 'N' Servers behind a Load Balancer
- (Y axis) Functional Decomposition
  - Microservices: split the 'dissimilar'
  - e.g. online-store (discussed earlier)
- (Z axis) Data Partitioning
  - Data sharding (segment the 'similar')
  - Logical separation of a domain.
  - e.g. Customer name, group based,..

Ensure the system is "just right" along the dimensions of Performance, Availability & Cost.

Y
Maximum Scalability
(Y) Microservices (component)
(Z) Data Partitioning (Sharding)
(X) Horizontal Scaling (cloning)
Monolith
X
Z
NPTEL
Data Center Networks
Advanced Computer Networks

Another fundamental principle that I want to present with respect to serverless is the traditional what we call as a Scale Cube. And this Scale Cube presents the three dimensions in which different applications can be scaled. If we consider x, y, and z as the three axes and consider the very base point with respect to what we see here as a Monolith as a complete entity that we have that is running.

And typically the way we scale them is like I said, we have to instantiate the replicas of that same instance to make sure that we are able to execute much more than what a single instance can offer. And this is what we call as horizontal scaling also called as cloning where the instance is cloned to run multiple of such copies.

And this duplicates when we build typically, we are trying to say that we are now having more processing capabilities and more performance that we can provide them what a single instance could do. Think of this as a traditional load balancer that we put and then scale the number of replica servers and all the services can be piped to say it goes to the load balancer, and you choose one of the instances to execute the actual function.

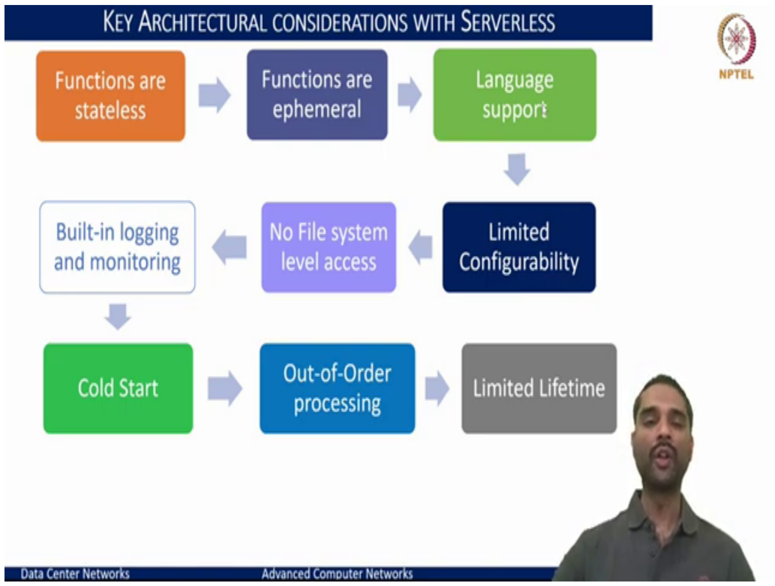
And this is a very common principle that is applied even in the Monolith design. The second scale, or the y-axis is where the microservices present us with the means to scale better. And this is possible when we do the functional decomposition of these Monoliths. And now, instead of scaling the entire replica, we get a choice to scale only those instances of the microservices which we need, really to push up the performance or pull the performance, and that is basically, if we disaggregate it into multiple dissimilar services, we can only scale one part of the instance be it just the databases that we want to scale, be it just the authentication service if there is a lot of cryptography that is slowing down the system or if it is a payment, you want to scale the payment gateways to ensure that we are able to do a lot of payments, which employ include multiple of the parties to work together. So, now, it gives us more handle in terms of which components is it that we want to scale and which components is it that we want to keep as it is or scale down as we go with respect to serving dissimilar traffic loads.

And the third important dimension in which we typically also scale is the data, essentially what the database we host for executing these functions. Here, the aspect is to take the data and shard it, wherein we are trying to segment the data based on the similarity on different aspects like if we say that we have a dictionary from A to Z, we can shard the data rather than having everything in one unit and access all the things from A to Z doing it in a single database, we can split it to say that A to E in one segment of the data are shaded, segment E to K maybe select a second segment, and K to S as a third segment and then S to Z as the last segment.

And based on our needs or wherever there can be places where things have a logical separation, we can split the data and make them accessible in different ways. So, that we are providing ability to basically partition the data and work with the data that we really see as a bottleneck that can be overcome pretty easily. And serverless tries to meet all of these three dimensions of scaling.

And hence, what serverless offers is what we call has the support for maximum scalability. And this way, we ensure that the system is just right about along all the dimensions of performance availability, making sure that you can scale up or scale out the instances and also making sure that you scale out just the right instances means you have a better light over the cost. And that is where the serverless has been gaining a lot of attraction and being a key propellant for this web applications.

(Refer Slide Time: 11:27)



So, having looked at the serverless model, in terms of how it is facilitating and what are the key architectural patterns that it governs on. To summarize in terms of what considerations come together when we want to think of serverless is to ensure that we want to build the functions that are essentially stateless. And these functions are not just stateless, but they execute for a short period of time; that is functions are ephemeral.

And functions when we say they are, in a way tied with respect to what kind of language we want. That means if we want these serverless functions to be much more diverse, we want to have good and rich language support for these functions. And by language support, we mean the runtimes that are necessary to build them.

And whenever we have any language support runtimes and build these functions, we also want to ensure that from a developer perspective, there should be minimal configuration overheads or the time that one needs to spend on our configuration should be very limited. And functions, if we think of them as just the execution unit is it also make it important for us to remember that we should not tie this up with any of the file system accesses because file APIs that will want to build routine have a tie to some of the system constraints, and better off is to execute these functions right from in-memory than having anything to do with storage, which will then add a lot of overheads together with the function. So, if you want to avoid such overheads on any constraints, it is better to keep in mind that we want to build functions that essentially have no file system.

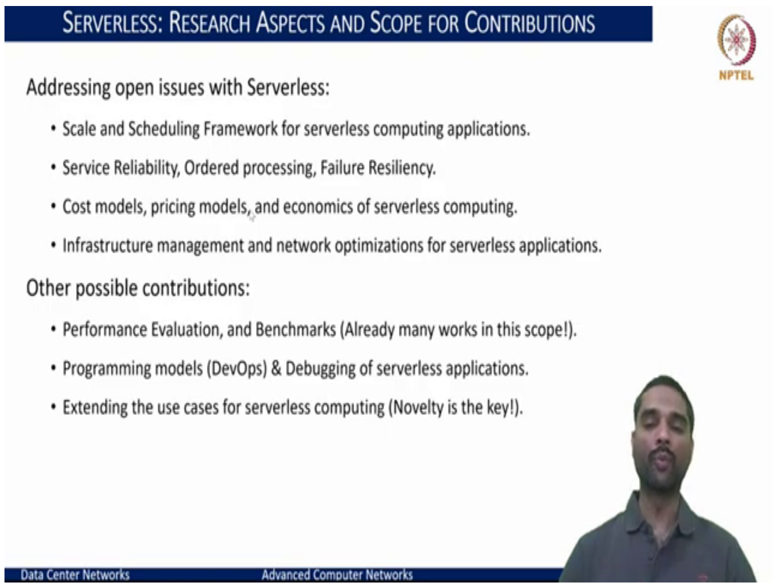
But rather can access the data through a a database or some DB APIs through which you can export and import the data in memory and then work with in-memory databases. And also like when these functions are going to be executed, we will also want to ensure that they have built-in logging and monitoring capabilities because it is hard to debug unless we have set the Screen works to log what is going on. And like we said also discussed earlier, in terms of the need to say which functions are executed and when and what are the impacts that they are going to have, we need a better monitoring framework to ensure that this all can be tracked.

And we will also speak about like what it means as a Cold Start, but it is essential that the functions, when they are going to be executed for the first time they would, encounter some overheads. And this results in what we call as a Cold Start, which may become a barrier for any latency-sensitive functions that we want to run.

And other important challenge that we will face is out-of-order processing, wherein if there are multiple events that are triggered, a function in a serverless context cannot guarantee anything with respect to the processing the events in order that is if event 1 was triggered before event 2, it does not guarantee that the function that is associated with event 1 will be processed the first then the event 2 function will be processed. So, there is no such guarantee that comes with these serverless functions.

And also the serverless functions would have to have a limited lifetime. They cannot be run forever. So, that means this also puts sort of lot of constraints in terms of what kind of language constructs that you would use, like we cannot have a while 1 or forever kind of loops to run and make these functions to work. So, we have to keep these considerations in mind when it comes to building the serverless framework.

(Refer Slide Time: 15:02)



Serverless: Research Aspects and Scope for Contributions

Addressing open issues with Serverless:

- Scale and Scheduling Framework for serverless computing applications.
- Service Reliability, Ordered processing, Failure Resiliency.
- Cost models, pricing models, and economics of serverless computing.
- Infrastructure management and network optimizations for serverless applications.

Other possible contributions:

- Performance Evaluation, and Benchmarks (Already many works in this scope!).
- Programming models (DevOps) & Debugging of serverless applications.
- Extending the use cases for serverless computing (Novelty is the key!).

Data Center Networks Advanced Computer Networks

And there is a lot of scope, when it comes to the serverless framework and how we can think of building better serverless functions and serverless frameworks themselves. First is in terms of like how we want to schedule the serverless functions, how we want to scale the serverless functions, the framework to ensure that the autoscaling can happen, and better kind of scheduling when you have lots of events to deal with and lots of functions, but limited resources on which you can schedule these functions. So, scaling and scheduling become important aspects to look at.

Second is the service reliability. That is, whenever the functions process and in between, if they fail, what happens to that request? What happens to that event for which the function was triggered to execute if it fails somewhere in between? So, we need to define what are the reliability requirements. How do we want to handle the scenarios when there are such failures? Do we want to retry? How many times should we retry? All of these aspects also need to be considered then in terms of the cost or the pricing models that tie in with the economy of serverless computing. How should we define the services when they are provided over the cloud? What should be the charging model? And should there be a means to compare the traditional model of how they are priced versus the serverless model? And we see that there are a lot of differences when it comes to the pricing models from different cloud service providers with respect to they offering the serverless function as a service model as opposed to infrastructure as a service or a platform as a service model.

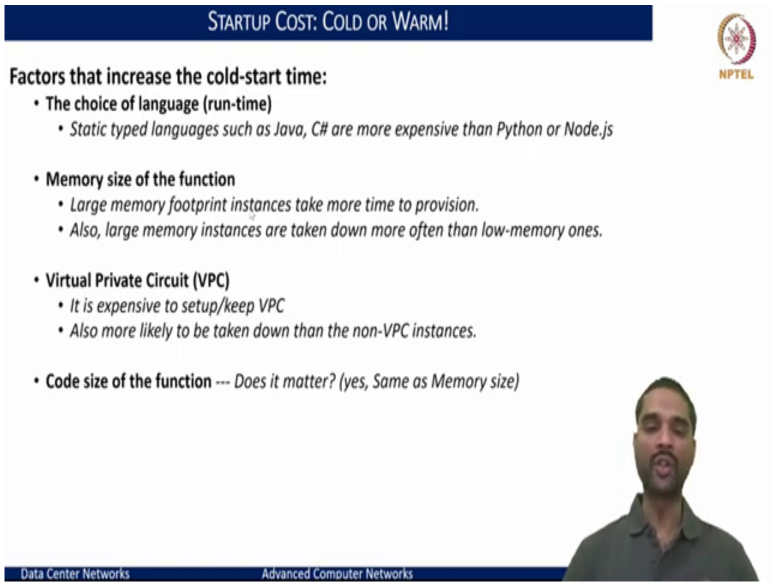
So, how justified are these pricing models? And what is the right pricing model that one should be catering to, all of these become important research aspects. And when we think from the cloud service provider perspective again, we have also see if I have to facilitate multi-tenant serverless functions, how do I manage the infrastructure for them? How do I ensure that there is an isolation that is guaranteed for each of the tenants? What are the service level requirements that we should be plumbing? What are the means to optimize on the network to execute that the serverless functions or applications can be done, all of these, again, become a very crucial aspect to think. And there are a lot more possibilities in terms of where the contributions need to be made.

And there were several of the researchers that have come in this direction over the last few years, including how to evaluate the performance of serverless functions, how do you benchmark the serverless functions, and say when public cloud-based serverless functions are better, when the other hosted private cloud-based serverless functions would work.

And even within the different serverless frameworks, whether Google is better or Amazon is better for you? What kind of workloads are each suited for all of these become important aspects to analyze? And likewise, if we want to see what kind of serverless functions that we would want to build and how do we want to debug the serverless applications, the tools around to build them would all be important contributions essential in this world.

And likewise, when we think of any application now, we can also think of saying whether it can be transformed into serverless computing and if so what kind of applications would really fit and any novel ideas that we can pitch forth in these areas will also become an important aspect.

(Refer Slide Time: 18:42)



STARTUP COST: COLD OR WARM!

Factors that increase the cold-start time:

- **The choice of language (run-time)**
  - *Static typed languages such as Java, C# are more expensive than Python or Node.js*
- **Memory size of the function**
  - *Large memory footprint instances take more time to provision.*
  - *Also, large memory instances are taken down more often than low-memory ones.*
- **Virtual Private Circuit (VPC)**
  - *It is expensive to setup/keep VPC*
  - *Also more likely to be taken down than the non-VPC instances.*
- **Code size of the function** --- *Does it matter? (yes, Same as Memory size)*

So, let us now try to look at some of the critical research concerns in terms of where the problems lie with the usage of serverless computing. And what are the key mechanisms that we might think of to overcome or provide better solutions to such problems and challenges in incorporating the serverless world. The first, like I mentioned, is the Cold Start or the Warm Start.

And what a Cold Start means and how this can influence we need to understand the key factors and what are the background things that are going when we want to execute a function and then try to see where the problems come. The first is to understand that for these functions it is essentially tied to a particular language. And the choice of the language has a great impact in terms of how the function itself can be deployed.

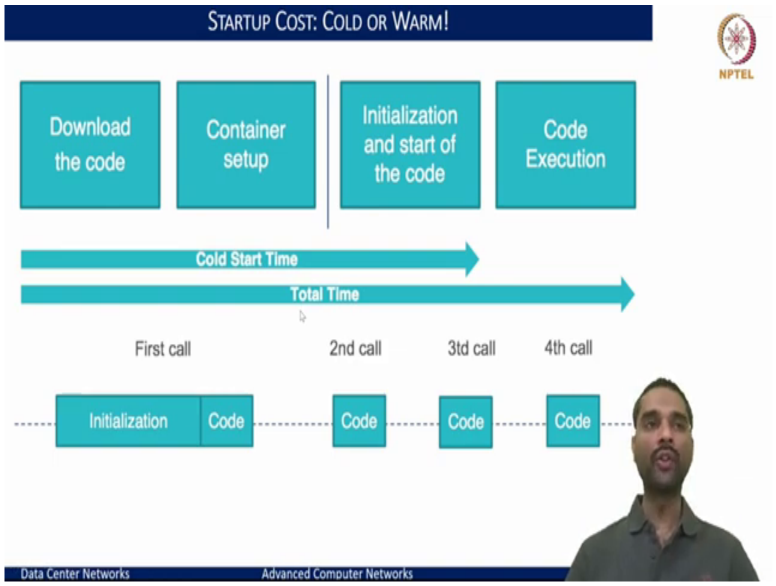
Like if we take any static typed languages such as Java or C#, they are going to take much more time in bringing up the essential runtime than, when we compare to the loosely typed functions languages such as Python and node.js, because the runtimes could vary and for a function to run, you need the runtime to be brought up, and if all runtimes require more out of memory data to be pulled in then they will definitely take a lot more time. And that is what happens with static typed languages. And likewise, when we have a function, and it is memory footprint is high, then it is going to take a lot more time to provision such a function. And if it all takes a lot more time to provision, that means you are going to endure a lot more time before you can really execute the function.

And second, often in the public clouds where you are sharing the resources, if I see a function that has a lot more memory and has been idle for some time, I would want to take it down so that it can make way for multiple other functions to run. That is where the large memory instances become the clear target members to be taken down whenever the system is experiencing a high load.

And likewise, is the code size as well, wherein if you have a large code size, that means you are going to have a larger footprint in terms of running this instance. And if you have such an instance, it becomes likely victim that was going to be taken off under those circumstances.

Also, the other aspect when we think of these functions is should we have support for VPCs or virtual private circuits because anything that you do in the functions to ensure that you have these private connections that is going to take a lot more of the run time, and it is also going to be computationally expensive to build those VPC pipes to ensure that the data can be passed through them in a secure manner. So, it is better off to avoid these VPCs when possible. And only for the functions which require to have this VPC, we should be thinking of building these VPCs as they are going to add a lot of time because before even you execute the function, before you can export or import the data. Your prerequisite becomes to set up these virtual private circuits and then only the actual data that can be moved in and moved out of these virtual functions. And these add to a lot of significant startup time.

(Refer Slide Time: 22:14)


Startup Cost: Cold or Warm!

So, to see what cold or warm start would really be, including whenever you want to execute a function, you will download the code and then set up the container with the necessary runtime and bring it up or initialize and then start the code to be run within the container. And only after these first three steps complete, we will be able to execute the actual function that is internet from the developer that is written as a function to be executed.

And the very first call, you have to do all of these first three steps that is booting up the Docker, setting up the runtime within the Docker, and then starting or executing the right process or the function that you would really want to execute. So, all of these take a lot amount of time. And that is called the cold start because you are starting completely fresh; nothing is there; you are creating a footprint for the first time and then executing the function.

So, if we look at these timelines, typically like, when we say the function execution times are very small, they have a range in the order of a few milliseconds. But when we think of the container boot up and then container setup in terms of the runtime, downloading of the code that we have into the container, and then starting the process, this takes in the order of hundreds of milliseconds.

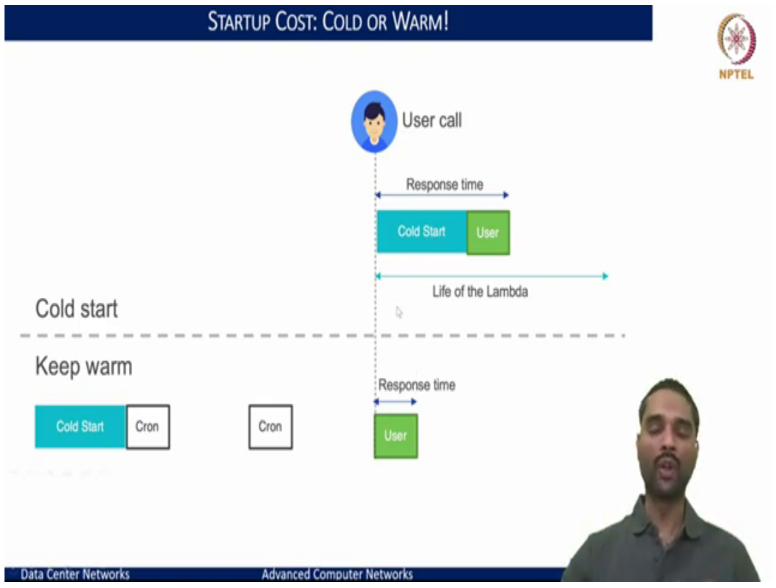
So, if we consider the very first call, our initialization phase consisting of this download code container setup and initialization and the start of the code would span for several 100 milliseconds while the execution of the code itself can be a few milliseconds. So, this is what we refer to as a cold start problem. But then, once this has been prepared, any subsequent call onto these functions would just take the time for the code execution.

So, subsequently, the second, third, or fourth on the same container instance would result in a very small execution time. And this is where we say the second, third, and fourth are already executing on a container that is warm, that is pre-built with all of these first three steps being complete. So, this is where we distinguish between a cold start and a warm start.

And now we can clearly see there is a need to cut down the time for the cold start. And then you have to devise the mechanisms that would enable us to basically cut down this cold start time. And where this really matters is with respect to what the response time the user sees. So, if it is having a cold start, the user would experience a very long response time whenever you send a request or whenever the event is triggered to the time that it takes for the response to come back is going to be large.

And this is the case with a cold start, while with a warm start from the user's point of view, you trigger an event and then the response comes back as soon as the function is executed. And this is what a user would typically be expecting. And if we have to have a rich user experience, we will want to be building our system with these response times, which are what a function execution will take, not the time that you would need to prepare such a container workload.

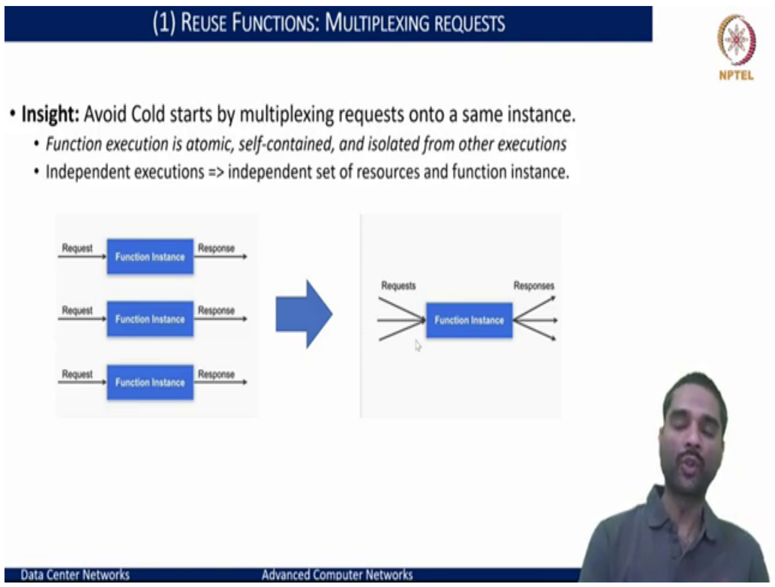
(Refer Slide Time: 24:30)



And now we can clearly see there is a need to cut down the time for the cold start. And then you have to devise the mechanisms that would enable us to basically cut down this cold start time. And where this really matters is with respect to what the response time the user sees. So, if it is having a cold start, the user would experience a very long response time whenever you send a request or whenever the event is triggered to the time that it takes for the response to come back is going to be large.

And this is the case with a cold start, while with a warm start from the user's point of view, you trigger an event, and then the response comes back as soon as the function is executed. And this is what a user would typically be expecting. And if we have to have a rich user experience, we will want to be building our system with these response times, which are what a function execution will take, not the time that you would need to prepare such a container workload.

(Refer Slide Time: 25:30)



(1) Reuse Functions: Multiplexing Requests

- **Insight:** Avoid Cold starts by multiplexing requests onto a same instance.
  - *Function execution is atomic, self-contained, and isolated from other executions*
  - Independent executions => independent set of resources and function instance.

Data Center Networks   Advanced Computer Networks

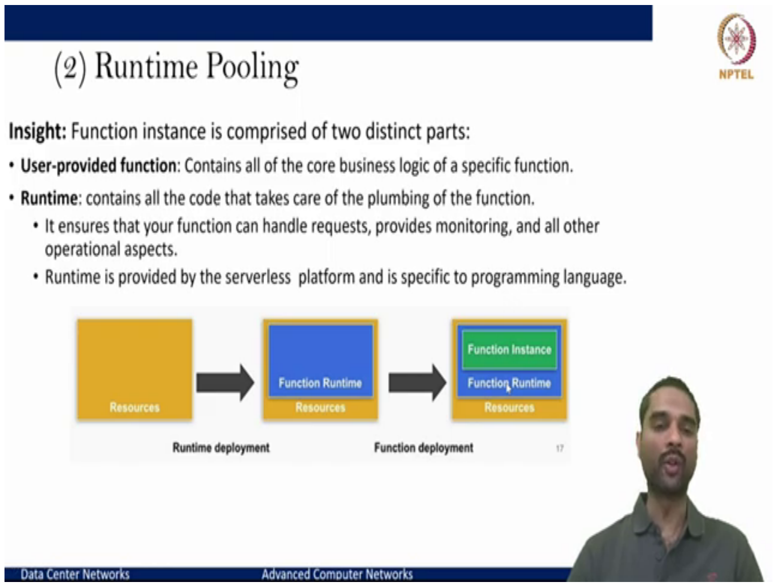
So, there have been several works that came up in terms of how to optimize and how to avoid these cold starts. Let us quickly rush through to understand some of the aspects that have been worked around for avoiding the cold start.

The first of them is what we call as Multiplexing the Requests. And the key insight here is that you can avoid the cold starts by multiplexing multiple of the requests onto the same instance like we saw in the earlier diagram, where once the first call is executed on an instance, subsequent second, third, and fourth calls were made on the same container.

And this ensures that you are able to do better. So, think of like if I have three requests, and each of the requests are going to be run on a different function instance then each of these function instances, 1, 2, and 3, would end up having a cold start and then execute the function and then provide the response.

Instead of that, if we had all of the requests to come on a single function instance, then only the first of the request would incur the cold start, and the remaining of the instances of what the requests that we want to process on this function instance would not incur any of the cold start overheads. Thus, reusing the container instance to execute these functions essentially helps avoid a cold start for subsequent of the requests.

(Refer Slide Time: 27:01)



(2) Runtime Pooling

Insight: Function instance is comprised of two distinct parts:

- **User-provided function**: Contains all of the core business logic of a specific function.
- **Runtime**: contains all the code that takes care of the plumbing of the function.
  - It ensures that your function can handle requests, provides monitoring, and all other operational aspects.
  - Runtime is provided by the serverless platform and is specific to programming language.

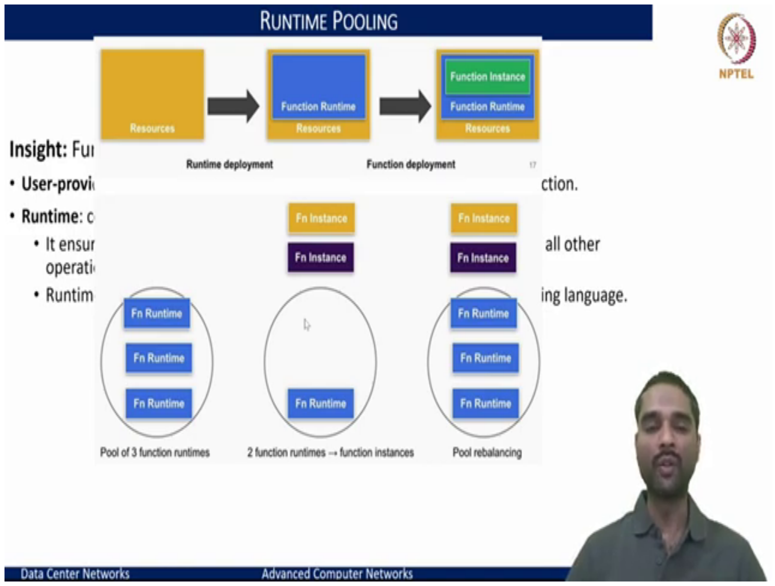
And the second approach that has been used is Runtime Pooling. That insight here is that if we have a function instance which basically comprises of the code that a user wants to do and the runtime framework, like if it is a Python, you need a Python runtime; for Java, you will need the Java runtime. So, the runtimes depend on the kind of function that we would want to build.

And if I as a cloud service provider, can create pre-warmed pools with specific runtimes. Like, if I am likely to execute a Java-based function, then I create a container with the Java runtime ready. So, then the only thing that I would have to do is pull the code that I want to execute and run it.

So, whatever the user-provided functions that we can think which essentially defined the core business logic is the only part of overhead that we should be paying, while the entire runtime for which your code needs to execute can all be pre-prepared and made sure that there is no overhead paid for generating a container, setting up the container and its runtime.

This way, we can have the function runtime and the resources for the container being set up apriori, and the only thing that would then have to do is just pull the function instance and execute. So, it would not cut down completely the cold start time, but it will take off the majority of the portion in terms of the resources and function run time which need to be set up.

(Refer Slide Time: 28:30)



RUNTIME POOLING

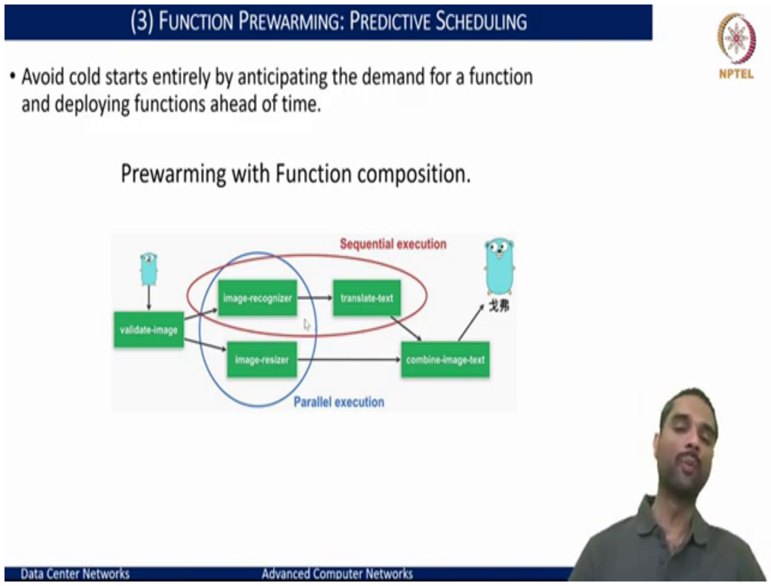
Insight: Fu ction.

- **User-provi**
- **Runtime:** c
  - It ensur all other operati
  - Runtim ing language.

Data Center Networks Advanced Computer Networks

Hence, this run-time pooling mechanism helps eliminate or minimize the cold start time. So, if you think of this, I can create multiple pools of function runtimes. And as and how I require, I can pull those into the containers that I might have and execute just the functions into those containers. And if we have created a pool, and we also have limited resources, we could basically rebalance our pool in terms of what functions are getting executed more frequently and then pull those function instances into a pool, which can then be deployed much faster.

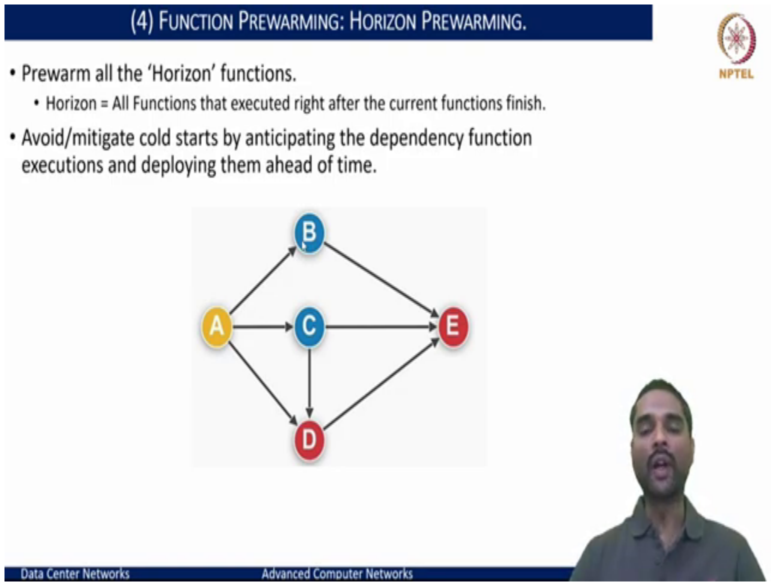
(Refer Slide Time: 29:04)



(3) Function Prewarming: Predictive Scheduling

- Avoid cold starts entirely by anticipating the demand for a function and deploying functions ahead of time.

Prewarming with Function composition.



(4) Function Prewarming: Horizon Prewarming.

- Prewarm all the 'Horizon' functions.
  - Horizon = All Functions that executed right after the current functions finish.
- Avoid/mitigate cold starts by anticipating the dependency function executions and deploying them ahead of time.

The third of the alternative is what we call function Prewarming, wherein if we anticipate that certain kinds of function will be executed in the near time, like when we see that it is in the morning hours, maybe will be interested in reading the newspapers, maybe if it is in the evening times maybe there is likely video streams, entertainment kind of activities or games, if there is any events that are going on. If we foresee that there is going to be traffic for a particular pattern, then we could predictively schedule those kinds of containers so that when the events come, the containers are all preset up, and you are only going to download the code and execute. We could even have the code instance that is going to be pre-prepared just that the event can trigger execution of such functions.

So, this way, we can avoid the cold starts entirely. But this anticipation also means it is going to come with a cost. And as we predict, our predictions may go wrong. And when they go wrong, it means that we may end up having to pay for the work that does not yield any value. But it is still okay if we want to know that there is a latency-sensitive functions that we will want to keep we can only prevent them, and any latency-agnostic functions need not be pre-warmed.

Further, it is very often shown that these functions often have a chain in terms of when I execute 1 instance of a function, and this is likely that this function is going to make a call into another instance. Like in this case, when you do a validate image as a function whenever an image is uploaded, this validate image as a function may call image recognizer or image resize.

And image recognizer may want to use the translate text service to prepare whatever is the extract of the subtitles or the text associated within the image. So, there can be a function that can yield into calling of other subsequent functions. And these can be done either in a sequential manner where you have a valid image, once you validate, you want to build the recognizer and then translate text, and so on.

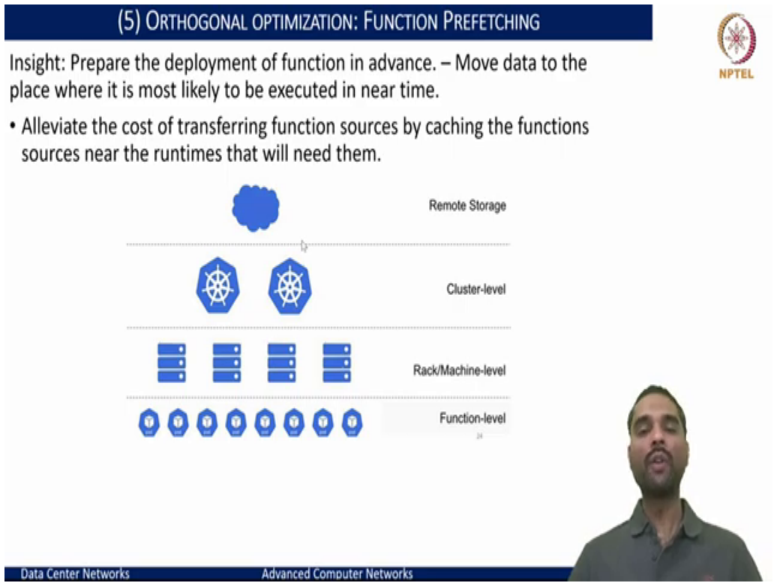
And also we can see there can be parallel executions of these pre-warmed functions in terms of when the validate image happens I want to run the resize service so that I can create multiple copies in different variants. And also, I can start to run an image recognizer. So, a single event can have multiple triggers and start multiple of the functions which can be executed parallelly as well. So, in either of the cases when there is a predictive scheduling that is being done, we would end up having a lot more benefit in avoiding the cold start completely.

The other alternative that is also being taught is in terms of what we call as a Horizon Prewarming. What it means is, when we say that there is a chain and I see that when I execute A, it is likely that the next time, we are going to execute our B, C, D and then followed by E.

Then, the moment that I am starting to execute A, I could pre-warm and bring up these B, C, D, and E functions and prepare the containers to be made ready to execute. And no matter whether A call ends up calling B, C, or D, we would have the instance up, and then we can execute the functions.

So, we can trace what would be the chain in which the functions are going to be executed and then have those functions ready as an anticipatory means to pull up and then start the execution. This way, we can avoid, if not completely cut down, but minimize the time that you would need for prewarming.

(Refer Slide Time: 32:47)



(5) ORTHOGONAL OPTIMIZATION: FUNCTION PREFETCHING

Insight: Prepare the deployment of function in advance. – Move data to the place where it is most likely to be executed in near time.

- Alleviate the cost of transferring function sources by caching the functions sources near the runtimes that will need them.

And an orthogonal optimization that we can think of is also Function Prefetching. And what this means is you prepare the deployment of the function well in advance. And this is, again, hinged on what we called as Predictive Scheduling. But here, the difference is we are trying to bring the entire function and keep it ready and expect that this function is going to be executed in the near time.

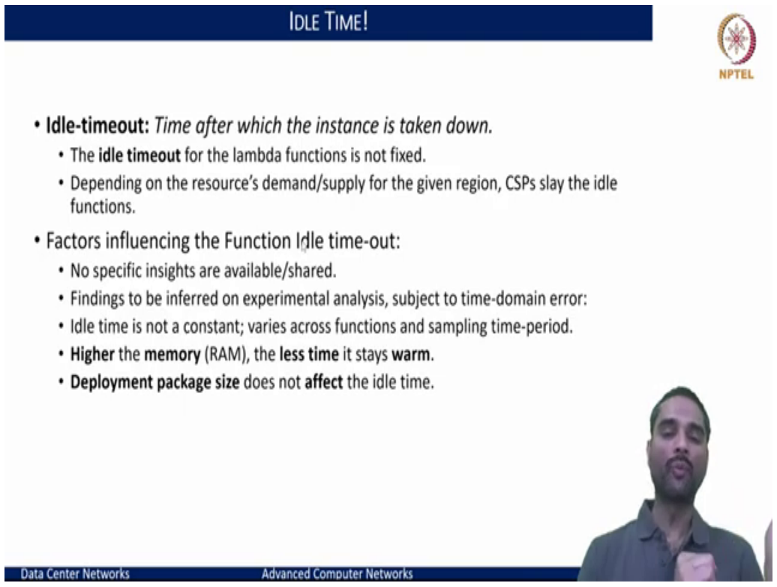
And this approach, in a way, can alleviate the cost of transferring the functions. That means if I say that in the next few minutes I have to execute a certain function I would want to keep that function as close to the instance where I would run as possible, which would make then that I can load up this instance much more rapidly I will not spend any overheads in transmitting a bulk of the data over the network.

And I can keep this data in memory as well, which ensures that I can bring up this function much quickly than going to connect and then get it from some external repository. So, these kinds of optimizations have been thought of and have been proposed in various means to say that we can avoid the prewarming.

And this completes one side of the story in terms of how the functions that we want to load, but then once we load, we execute what do we do with it? Should we keep them waiting so that we anticipate the next request will come and I can reuse the function instance? Or should we take them off and execute a new instance with a new model?

If we follow the latter, then we are likely to hit the problem of cold start issues, having repeating multiple times. And if we want to stop them, then we do not want to be taken down the functions quickly as well then what that would mean is a function once it is brought up, it is going to have some idle time.

(Refer Slide Time: 34:52)



IDLE TIME!

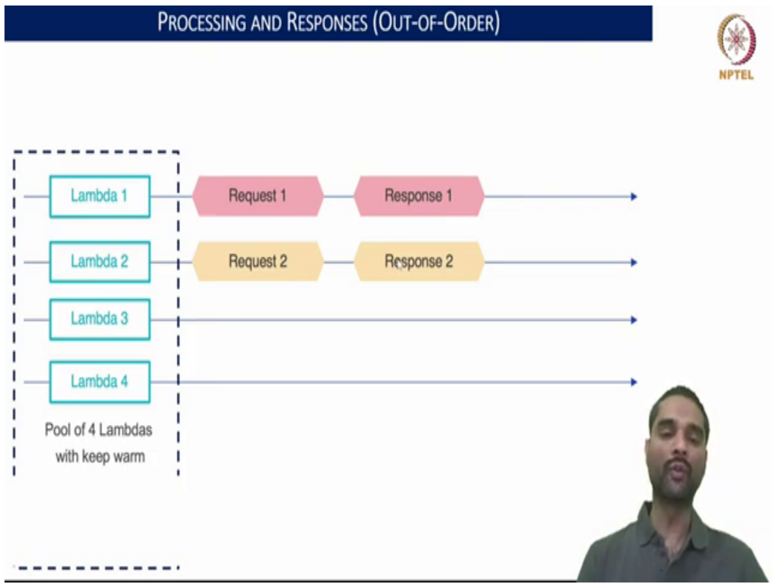
- **Idle-timeout:** *Time after which the instance is taken down.*
  - The **idle timeout** for the lambda functions is not fixed.
  - Depending on the resource's demand/supply for the given region, CSPs slay the idle functions.
- Factors influencing the Function Idle time-out:
  - No specific insights are available/shared.
  - Findings to be inferred on experimental analysis, subject to time-domain error:
  - Idle time is not a constant; varies across functions and sampling time-period.
  - **Higher** the **memory** (RAM), the **less time** it stays **warm.**
  - **Deployment package size** does not **affect** the idle time.

Data Center Networks Advanced Computer Networks

So, how do we keep this idle time? Should we have a timeout value based on certain characteristics? And if we want, how long should we wait before we timeout and then take the instance down? All of these are, again, crucial factors with respect to how we want to deploy these serverless functions. And this idle timeout need not be fixed because some functions would want to run for minutes, but they may run hundreds of requests in a minute and then go away. But some may serve just 10 or 20 of the requests, but they may run for a few minutes. So, they may vary in terms of what the requirements could be. So, the idle timeout again, it is hard to define one value as a thing that will work for all, but we need to be more clear in terms of what these values need to be.

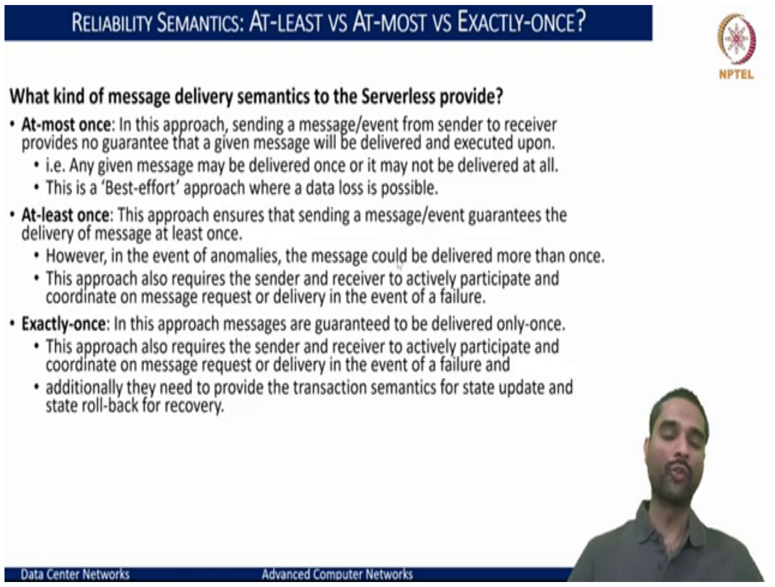
And this also means if we have to be more subjective, we want to ensure that we have the expertise or we have experimental analysis that can help us build and define what should be the idle time. And the problem with the idle time is that we are going to eat out the memory, eat out the resources on the devices although we stay idle, but the only benefit that we would have is any requests that come in the meantime would not have any cold start problem but just execute in a warm start model having just the response time dictated by the execution time of the function rather than any of the overheads and this is where again several of the concerns in terms of which instances need to be kept up which instances need to be taken down. And that depends a whole lot on the memory size and the code size and what are the overheads they bring.

(Refer Slide Time: 36:31)


Processing and Responses (Out-of-Order)

And the other challenge that we have with respect to the functions is the out-of-order processing because the requests can be spawned at different times, but the sequence of the requests or the sequence of the responses need not necessarily match. So, once we have requests that are going to different lambda instances or different functions in this case, we may start to see that request 1 that is going on lambda 1 may take a lot more time for a cold start to bring up or because of the resource constraints on which kind of hardware that it is going to run or because of the pools that would have already prepared there may be a function instance that is available in a pool there may be an instance that is not available. So, all of these factors can play a different role in how the latency is affected for response. And we may say that even if request 1 and request 2 were sent in sequence, we may end up seeing response 2 first before response 1. And hence, there cannot be any guarantees in terms of the order in which the requests are going to be processed in a serverless model.

(Refer Slide Time: 37:39)



RELIABILITY SEMANTICS: AT-LEAST VS AT-MOST VS EXACTLY-ONCE?

What kind of message delivery semantics to the Serverless provide?

- **At-most once**: In this approach, sending a message/event from sender to receiver provides no guarantee that a given message will be delivered and executed upon.
  - i.e. Any given message may be delivered once or it may not be delivered at all.
  - This is a 'Best-effort' approach where a data loss is possible.
- **At-least once**: This approach ensures that sending a message/event guarantees the delivery of message at least once.
  - However, in the event of anomalies, the message could be delivered more than once.
  - This approach also requires the sender and receiver to actively participate and coordinate on message request or delivery in the event of a failure.
- **Exactly-once**: In this approach messages are guaranteed to be delivered only-once.
  - This approach also requires the sender and receiver to actively participate and coordinate on message request or delivery in the event of a failure and
  - additionally they need to provide the transaction semantics for state update and state roll-back for recovery.

Data Center Networks Advanced Computer Networks

And this also has a very important bearing in terms of how often would a request result in execution. And especially when we have a network system which is prone to failure, what happens to my request if the instance fails and there are 3 models of executions? One is called At Least Once Delivery Semantics, and the other the At Most Once Delivery Semantics and then Exactly Once Delivery Semantics in the network systems.

And now, when we made a serverless framework or platform, what kind of a model or semantics should they be supporting becomes an important question. And the most common one is basically At Most Once what it means is this is the best effort model wherein once the event is triggered or message is being triggered from sender and receiver, it will try to just have the best effort in ensuring that it is executed.

If, for some reason, the message is dropped or is not delivered to the instance or it is delivered to the instance but the instance, while trying to execute, fails for some reason, then there will be no retries that are going to be done, and no guarantees are going to be provided in terms of whether this execution completes or not.

And hence this is similar to the best-effort IP of the network model that we often speak of where if there is a data loss, you give up. The other alternative I think of is the At Least Once so that as a sender when I send and I do not get a response in time, I retry like how we do with TCP so that you retry for certain times and see whether things succeed and you get a response back. You can keep retrying until you get a response. And then this mode of semantics means that I am trying to send the requests multiple times for the receiver to take it and process. It could happen that the receiver has already processed, but before he could deliver the response back, he might have crashed, or the response itself was lost. And in such cases, you would end up seeing that the execution of a function would be run multiple times.

So, what this model essentially guarantees is if there is an event that is being triggered at a function that needs to be executed, it will be executed at least once. And it is also likely that in trying to make some retries, we may have executed this function instance more than once. And that is why the semantics is called at least once.

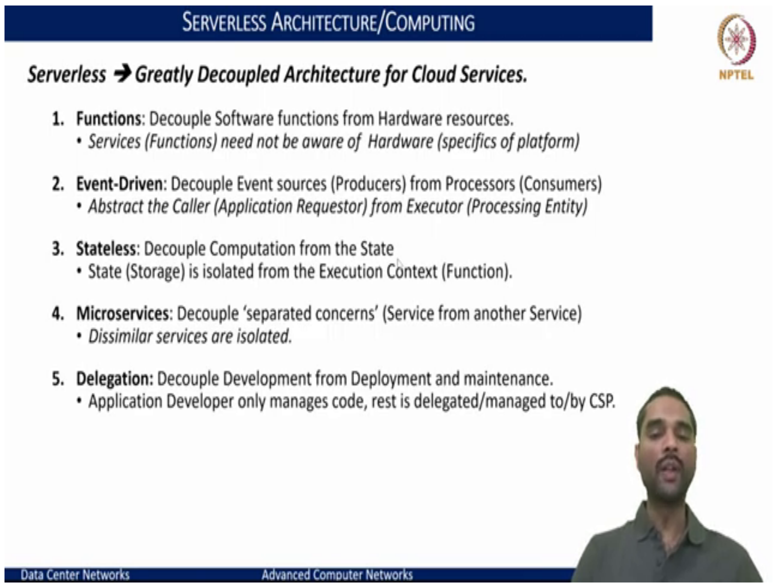
And definitely, we can see that this would not be the right fit if we are talking of any financial transactions that we have; neither would the At Most Once model work, nor would the At Least Once if we are having like banking transactions where we credit or debit the amount in our bank accounts or even if you are trying to do share trades or whatever, you cannot have the best effort wherein we try but nothing works nor should we have that I debit once and it shows up as debit multiple times.

So, these two models are good, where it is non-critical services that could be run. But for critical services, what we need is the Exactly Once Model. And what this ensures is once I do a transaction or once I send a message, it would ensure both between the sender and receiver that there is only one time a function would execute and a response would be delivered back to the sender.

And even in any case of failures, we will want to have the means to recover from the failure and ensure to see the precise state where the failures occurred, continue from there and ensure that the function executions happen just once. And this is very hard to meet, and serverless frameworks, most of the ones that we see are providing either the At most Once or At Least Once, like the Google Cloud, Amazon Lambda, all of these services typically cater to the At Most or At Least Once and not the Exactly Once pattern.

Nonetheless, this would be a great feature that is desired. And how to make it happen is one of the open research challenges. And especially to make it happen in an efficient manner without having much of overheads is the key. And this is where a lot of research attention has also been over the last few years.

(Refer Slide Time: 42:00)



SERVERLESS ARCHITECTURE/COMPUTING

Serverless → Greatly Decoupled Architecture for Cloud Services.

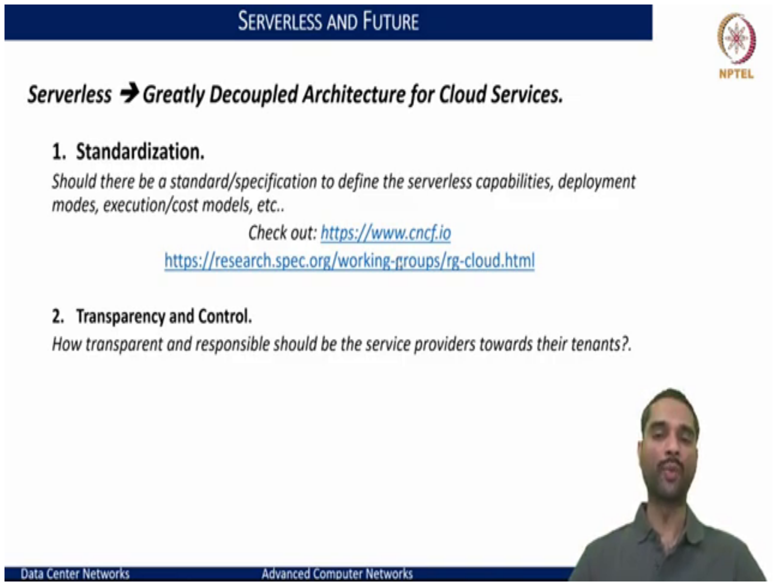
1. **Functions**: Decouple Software functions from Hardware resources.
   - *Services (Functions) need not be aware of Hardware (specifics of platform)*
2. **Event-Driven**: Decouple Event sources (Producers) from Processors (Consumers)
   - *Abstract the Caller (Application Requestor) from Executor (Processing Entity)*
3. **Stateless**: Decouple Computation from the State
   - State (Storage) is isolated from the Execution Context (Function).
4. **Microservices**: Decouple 'separated concerns' (Service from another Service)
   - *Dissimilar services are isolated.*
5. **Delegation:** Decouple Development from Deployment and maintenance.
   - Application Developer only manages code, rest is delegated/managed to/by CSP.

Data Center Networks Advanced Computer Networks

So, to sum up, we have looked at serverless, which is basically a greatly decoupled architecture and serverless. The key thing to remember is that it is all about the functions that we want to execute or function as a service on the cloud model. And it is driven by the architecture pattern, what we call as event-driven architecture.

And the functions that we want to build here are stateless and leverage the microservices pattern where we have separated concerns. And also builds on the model of delegation wherein the developers would be completely decoupled from the maintenance and management aspects of the functions.

(Refer Slide Time: 42:42)



Serverless and Future

Serverless ➔ Greatly Decoupled Architecture for Cloud Services.

1. **Standardization.**
   Should there be a standard/specification to define the serverless capabilities, deployment modes, execution/cost models, etc..
   Check out: https://www.cncf.io
   https://research.spec.org/working-groups/rg-cloud.html

2. **Transparency and Control.**
   How transparent and responsible should be the service providers towards their tenants?.

Data Center Networks   Advanced Computer Networks

So, to think of serverless, think of it as a greatly decoupled architecture. And other aspects to think are the standardization in terms of how would we bring forth the standard specifications to define the serverless capabilities, ensure that they can be deployed wherein you want to facilitate interoperability of the functions that you deploy on one serverless cloud service provider to the other pattern all of these have to be worked out.

And there is a lot of research that is going on in the cloud working groups like the links that have provided here would throw more light in terms of what are all the challenges and how people as a community are thinking together to build this and standardize the serverless frameworks. And the second is also in terms of transparency and control, that is, how transparent and responsible should be the service provider towards their tenants.

Because it cannot keep the tenants in the dark and say I manage everything, and it is my responsibility how I do without offering any service level guarantees. So, what level of transparency needs to be brought so that the service providers are able to offer the tenants with the necessary service level agreements, and how they are able to even deliver that these parameters are being met? How do they provide the statistics, the responses back all of these would be important aspects to look at.