Changwon:

Hi, everyone. Thanks for the introduction. My name is Changwon and I work as a DevOps engineer for Devsisters, and I have a lot of things I want to talk about. But before I start, how many of you know what CookieRun is? Great. And how many of you know that there is a new CookieRun game coming up in two weeks and have preregistered for it? Not so many of you? That's why I brought this QR code with me. So do preregister with a QR code, and you'll get this gift I brought from my office. It's cute, right? So I'm just going to let this guy sit by.

And so while you guys are doing what you have to do, Devsisters is a South Korea-based mobile game developing company. We developed the CookieRun franchise mobile games. I brought up four that's relevant to RoachFest. And we published these games ourselves, except whenever we cannot due to regional legislative issues or there's a perfect match between a publisher and the game itself. And these games, the CookieRun franchise has over 200 million users worldwide.

So a little bit about my team. My team is a central DevOps team within the Publishing Platform Group. So our team doesn't really develop the games ourselves, but instead we oversee a DevOps SRE culture and practice. We do infrastructure provisioning and a little bit of platform engineering. So every launch that happens within our company has to go through us, so we have accrued quite a lot of launch experiences that we're able to provide some software architectural advices and consultations on what's going to happen on launch and what needs to be considered and so on. And we also implement some abstraction layers and guardrails for fail-safe infrastructure provisioning as well.

So being a central DevOps team in a multi-studio game developing company means that these game studios bring up with their own unique and varied needs. And of course, directly fulfilling each of those requirements is, of course, practically impossible because we're basically outnumbered. So our approach to this mission is to basically consolidate all these requirements into one robust solution that we can work on, using the basic building blocks of cloud infrastructure, Kubernetes, and Helm charts, which I think are basically the Latin language of DevOps field these days.

So these are the four CookieRun games that are basically powered by, or will be powered, by CockroachDB. CookieRun: Kingdom was released in 2021. It was the first title for us to use CRDB and goes beyond. And we have successfully launched last December in China with Changyou and Tencent as our publishing partners. CookieRun is actually the first game that went public, and it got some attention. It was launched in 2013, and Cockroach didn't exist back then. So we started from Couchbase, and we are actually in the progress of migrating our database to CockroachDB because we're preparing to launch in India, because India is an emerging mobile market these days. And we're developing talks in technical details with Krafton.

Witch's Castle is a tap-to-blast puzzle genre, and it's doing great in Asian countries on its own. Tower of Adventure, it's coming soon on June 25th. It's the one you preregistered for. And we are planning to launch globally except for Japan because for Japan, we're going to be working with Yostar to be our publisher.

So that's already four different game studios and three different publishing companies, of course outside of our organization, that our team has to work on. That sounds insane, right? But we believe that we have everything handled and we can make things happen in orderly manner. And I'm going to explain to you how.

So that's the plan of the day. I'm done with the introduction, and I'm going to run briefly on why we use CockroachDB and throw out some tips in operating CockroachDB. And I want to show you how we actually deploy CockroachDB and showing some of our codes, but I might have to skip through that part.

So why we choose CockroachDB. We actually have a case study published online. So it contains all the details, but it all comes down to these points. So before, we were using a document-based database because in gaming, things change a lot. We have frequently-changing schemas because we have to update and add new features in on monthly basis. And changing schemas in database terms means schema operations. And we didn't want to deal with that, so we obviously went for document-based databases. And what was neat about that is that they often come with distributed horizontal scalability.

But that was years ago. The market's evolving. Our users are evolving, so people were asking for more interactions between each other, things like they fight against a raid boss with other players. They want to form a community within the game, like guild features. So all these kind of features require transactional support.

And also, we have seen with our own eyes that we sometimes have to face extremely heavy traffic. So we didn't want to lose out on the horizontal scalability part, horizontally scale out if there's heavy traffic, but also scale in as well because mobile games tend to stabilize downwards because it's a competitive market. So we have to have horizontal scalability as well. And we always wanted to choose consistency over availability, because even if... We don't deal with actual money. It's the people's time, money, love, and affection they put in our games. So every progress they make in our game, we wanted to be there. Things have to be intact.

So we did some research, and this is probably the matrix that you have all internally within your organization. And you thought CockroachCB either stood out or they deserve a chance, so that's why probably you're here. So I'm not going to go through the numbers. They're back from 2020. But that's what we did, and here we are.

So we are running pretty much everything on Kubernetes, and we didn't want to make Cockroach an exception. So we are also running Cockroach on EKS. We are also running the Cockroach nodes on a dedicated nodes pattern, meaning that they have their own resources because database is an important workload. We don't want to make them compete for resources with other workloads, except for things like the DaemonSets and things like that. And we also have multi-AZ setup for these Kubernetes cluster... I mean, the Cockroach clusters.

And for monitoring, we use Datadog as our monitoring platform. And Datadog has its own AWS and CockroachDB integrations so that you can work with metrics to build dashboards and monitoring alerts. But this can be done with Prometheus as you know, so you can choose whatever you prefer.

The key metrics that we keep eyes on are USE metrics. So that's utilization, saturation, and error rates. That is to discover what's going on at host machine levels. And we run everything on EBS. We used to use local SSD devices, but you don't want to go down that path. So trust me on that. The EBS, of course you have to look at the IOPS/Throughput on the block devices, the Queue Length and Write Stalls, all those EBS-specific metrics. And for CockroachDB, we look at all the other metrics as well. But these two unavailable ranges and capacity, as in storage, are the two metrics that we mainly focus on because that's where the data losses may occur.

And if you run a self-hosted Cockroach cluster, you sometimes have to replace the pods from time and time because you have to upgrade the Kubernetes version, you have to upgrade the CRDB version, and there are instant retirement things. And we figured we can do these kind of operations without any downtime unless you go too fast, because we had an incident where we tried to decommission something like seven to eight nodes at a time from an 18-node AZ. And ever since then, we just decided to roll out only three nodes at a time max in a same AZ because, of course, to avoid any data losses.

And we also have the rebalancing rate limits adjusted from the default value to happen the data transfer, and things happen quickly because we are Korean teams, and Koreans like to do things [foreign language

00:09:43], which means quick and quick. So we use the values 128 megabytes to be optimal. If you go beyond that, the network transfer and disk I/Os that happen because of rebalancing may interfere with the actual workload and cause the product workload to be slow or goes on and off.

And since we are running everything on EBS, there are EBS hiccups because we are living in a physical world. EBS has four nine SLAs and it's designed to be highly durable. So it has a primary and replica set up for the volume devices under the hood, and there are hardware failures on those volume devices. And whenever the primary volume faces one of these machine failures, then it will quickly turn over to its replicas. And I/O does pause when these failovers happen. And if it's running a CRDB node on these EBS faulty volume device, then the CRDB node crashes and restarts on its own.

So that's the log that we took from our cluster saying, "Slowness detected and fatal error can occur due to faulty hardware things." And oddly enough, we were seeing unavailable ranges happen when we see these kind of logs, which is kind of odd because for unavailable ranges to happen, you need to have this event occur over different AZs, which is statistically very unlikely. So I really think it should be under replicated ranges, but I don't know why. And our AWS support NSA couldn't figure out and link with what's happening on the CRDB. So just decided to move on because it wasn't giving any service outage for us, but I didn't want to sound too assuring because it may depend on the workloads that you're running.

The machine control has been really nice for us. We have analytics workloads that basically runs ETL tasks on a daily basis. Basically, what it does is that it has to capture the state of the users at certain time of the day, and there are a very high surge of SELECT queries at this point. But our cluster was just the right size for the workload itself and not for the ETL tasks. It was like two to three times more than what we're expecting, and it's not something that we want to invest in for just a 30-minute window of the day.

So instead, we set the session level QoS for the OLAP queries, and that's exactly what the admission control is for. Then the CockroachDB will prioritize other workloads over these QoS ETL queries. So that's what happened. We were seeing somewhere like two seconds P95 query latencies. After we introduced admission control, it reduced down to a hundred milliseconds.

And I want to put a personal emphasis on this slide. We're all human beings, and human beings make mistakes. It's easy to accidentally delete resources that shouldn't be deleted in Kubernetes, so I highly recommend that you introduce a fail-preventing webhook. You can use Kubernetes admission control to block whatever that shouldn't happen, like we wrote our webhook to deny any delete requests on things like StatefulSet or persistent volume claims unless a specific annotation is set. So you can see that the admission webhook, delete protection webhook, is complaining about the delete allowed to be true to require for it to delete the StatefulSet.

So you could write this using the boilerplate, but you would have to work with the Go projects and build it, build an image, and deploy it, which kind of feels like a tool to us. So we just wrote a simple tool that lets you write these kind of webhooks in JavaScript and deploy it. So it's open source and you can go check it out. It's called Checkpoint.

I put a personal emphasis on it because there was an incident caused by myself. I feel like I became a person who talks about their own incidents every time I come to RoachFest, but here it is. And we have a postmortem document that's written in Korean, but I figured K-pop wasn't as big of a thing as to a point that you all felt the need to learn Korean so I had ChatGPT translate the document for us and here it is.

So basically, what happened is that I accidentally deleted the Argo application that one of the CRDB AZ was running on. So basically, what happened was I was running some tests because we were transitioning the CPU architecture from AMD64 to ARM64. We are running on all Graviton these days now. So it's a small change, but it's one of the biggest changes in computer science as you know, so I thought it was worth a test.

So I was done for the day and I decided to tear down the test environment. So I went into browser and typed ArgoCD, but the thing was that I got back on that day from a Pilates session and I got extremely exhausted, so tired me typed ArgoCD and the browser auto-completed a CRDB application. And having worked for the testing environment for the day, I automatically thought that I was working on a test environment. In fact, of course as you might know, it turned out to be the prod and I pressed delete. And of course, ArgoCD has these confirmation prompts, but you just copy paste whatever's written on the prompt to the input. So that doesn't really stop you from actually deleting the resources. So that's what it goes.

But the good part was... Everything start to disappear, and we have our servers configured so that it talks to the database on the same AZ to not use the data transfer. So I was seeing, all of a sudden I was being paged with the server errors. And I was like, "I was working on a staging environment." Two seconds later, I realized what's happening and I was like, "Oh."

So thankfully for me, we had the delete protection webhook deployed to all of our clusters and it prevented the StatefulSet and the PVC from being deleted and these kind of deleted the ArgoCD application as well from being deleted and it had all the data intact. So all I had to do is to reinstate the ArgoCD application, and everything went back to normal. So these kind of webhooks not only prevent from human errors, but also prevents you from being fired and lets you even talk about these on this event.

So before I jump into show you how we deploy things, I want to describe some of the characteristics of mobile game development. If you ask me why our company exists and I would say in one word, it would be to entertain the users. And to entertain our users, we frequently deliver a lot of creativity because mobile game requires to be creative. I mean, it's got to be more creative because compared to other gaming industries, you didn't bring any gaming consoles and probably brought some PCs, but they're not for gaming, right? But you do have a phone, right? As a famous game developer once said, "You'll have your phones and basically you're more frequently engaged. You're likely to be more engaged with your mobile games." So the contents are depleted at a faster pace, and we have to deliver a lot of things at a frequency.

So all that leads to us is to work on multiple features in parallel. So each engineer has to work on their own features in their own branches. But the thing is that with the parallel development, as you all know, that it may break things around. It may not be compatible. It all has to be merged and QA'd at the end of the day to be delivered. What's worse` for us is that we even need the multiple production environments for the actual prod and the app store reviews and press events and onsite game shows. So the developers need to basically spin up the isolated server environments. So they need to kind of stamp out the server deployments as they wanted just like this, which is stamping our cookie characters taken from one of our trailer to Witch's Castle. I thought it was a nice metaphor.

So we came up with this concept called serverfarms. It's defined to be an isolated server applications that are reproducible and fully operational by itself. And the technical requirements for this are reproducible, that it should be declaratively defined so that the end results are isomorphic. It has to be isolated, meaning that no data or network traffic should flow in between or depend on other serverfarms. And also, that there has to be self-service because developers working on their own branches should be able to configure for their branches and deploy it whenever they're ready to test their new functions.

So basically, this is the topology. This is what happens. So the engineers provision one of the Serverfarms and the QA engineer and the server and client devs have their look at their own serverfarms. And even the app store reviewers got their own serverfarm because they're probably working on an unreleased feature.

So back in the day, like five to 10 years ago, these were a terrible set of requirements to fulfill. But Kubernetes, I think, is actually designed to meet, to solve these problems really neatly. Helm charts, by definition, are reproducible and declarative. We have Kubernetes namespaces that can logically isolate each workloads. And for the self-service part, we just wrote a light web application that basically runs Helm charts and lets you monitor what's going on in the Kubernetes clusters.

So that's what the light web application looks like. So you can name the serverfarm and you can set server versions for different server components. And here next are the status and whatever that's been deployed on the Kubernetes. And it's at the actual Kubernetes namespace that's been displayed. I mean, it should really be the same names, but I took these screenshots from a different point in time so they don't match but you can imagine that. We also have CockroachDB deployed as a part of our Helm chart. CockroachDB manages the official Helm chart and the benefit of using CockroachDB as a Helm chart is of course, it's flexible, it's automated, and it's reusable. And I'm going to cover it one by one.

So this is basically the chart manifest that we have that's been using for test environments and it's going to be live in production in the next two weeks. So we have this CockroachDB included as a dependency on the second block. We might be pined on the version, but that's what we happen to use and we are happy with it. And you can basically spin up a functional CockroachDB cluster just by doing that. But the whole point of having a serverfarm was so that it can be fully functional by itself.

Having a cluster functional doesn't mean the application to be ready by itself. You need to initialize the database with some schemas and inject with some cluster settings and things. For that, we use Kubernetes Jobs. Actually, it's taken and expanded from the official Helm chart. So basically what we do is that we initialize a cluster, we integrate the OIDC with our identity provider. We do all these one-off bootstrapping jobs, like applying the CRDB cluster settings, creating backup jobs, creating users, roles and granting privileges. And of course, there are some components that are just way better off when you use managed services like Redis. You don't want to deal with it in Kubernetes because they're highly volatile.

And there are some environments like PROD and stage, that are defined to be more rigid than DEV environments where things need to be ad-hoc. For those, we use Terraform. So basically we built a module that when you apply, the entire serverfarm appears using the cloud resources and it deploys things on Kubernetes. Of course, you can see that Redis and [inaudible 00:23:30] appear on the list of files, and we deploy the application Helm chart and the CockroachDB Helm chart. And what's neat about this is if you use the Kubernetes providers, then it makes this module completely agnostic of the cloud provider and the regions.

Of course, you would have to work with the Redis and the object storage and things, but every cloud provider has its own resource, so you can easily migrate when you decide to change the cloud provider. We have the CockroachDB block just right there as a Terraform attribute. And basically what it means is that it kind of transforms into an ArgoCD application, which then installs the Cockroach Helm chart with the values defined below. And basically, that's what I deleted just a few slides ago. This might have sounded boring and I hope actually it did sound boring to you, because the whole point was that we shouldn't invent the wheel.

And we kind of internalized this principle as a dogma within our team, when we were working with Tencent, who is obviously outside of our team. It is often an engineer's dilemma that we do have a tendency to solve problems in fancy and complex ways. We want to think of brilliant ideas, but these brilliant ideas need to be convinced to others unless the ideas speak for itself. So we should remember that complexity always comes with a cost of learning and communication. And there's simple and straightforward solutions. It will help you to easily collaborate in an efficient communication manner, especially when you're working with organizations or teams outside of your boundary.

So these are the main four tools that we use. Terraform, Kubernetes, Helm, ArgoCD, also known as Plane, right? So these are the points I made throughout my presentation. We run everything on Kubernetes. We use Kubernetes as a baseline of our infrastructure deployment, and that allows us to be agnostic of cloud providers as well as regions. And if you're going to agree with us, do make an error-preventing Kubernetes webhook because it's going to save your day one day. And we came up with this abstraction of application development using the very common tools, Helm charts, Terraform and ArgoCD. And that's all been possible because CockroachDB is designed to fit naturally in the cloud native operations.

So just a final note, make everything boring to easily communicate with other teams, especially because you get bored when there's nothing new and when you're not intellectually intrigued. But that kind of means that you don't have to explain further. And the idea is you don't require further documentation, comments and things like that to explain what's happening on your infrastructure deployment. So that's been our secret in deploying things in different cloud providers and other regions. So I hope it was worth your time, and here is our game that's coming out in literally two weeks. So please, try it out. It's going to be fun. Try it with your friends, kids, whatever. Yeah, and that's it. Thank you.