How Dream11 hit a peak traffic from 1200tps to 12k tps to 48k tps to 400k tps in subsequent years of IPL and how were they able to create a Guinness world record for largest no. of contest joins in a single match and whey they don't make use of Amazon ASGs and Google Cloud Analytics and why they shifted from monolithic approach to microservices with increasing scale?

Dream11 : Scaling to serve 100+ million users

Amit Sharma is the CTO of dream 11. He earlier worked at Yahoo, U.S and Netflix that's when Netflix also made a journey from monolithic to microservices.

Amit explained the challenges they faced every year for the scale they've now at Dream11.

They faced different issues in various years as they've progressed and the things that mattered at a time for them , no longer they considered them in the future.

Their priority is always open source, of-course to save cost and share knowledge.

Unlike other platforms their traffic is very seasonal in nature considering that they're heavily involved in nature.

From ipl to ipl, there is a surge in the number of users that come into the user and traffic they get.

At fantasy sports, we select a real world match be it cricket or football , using players of 2 teams we create a team of our own using certain constraints and combinations. And once the match actually starts and players start performing. Based on their performers, users start getting points. Now, there will be total number of points user team will be accumulating and now users will compete with each others team. 2 or more users can contest (free/paid) with one another.

Back in 2016, they were very excited. They hit a peak traffic of 1200 tps or rps which is a very healthy number.

Then they started to prepare for the next ipl. At the next ipl, they went from 1200 rps to 12000rps. That was like a 10x Growth they actually received.

They were preparing for that and built a good infrastructure for that.

The traffic went from 12k to 48k rps in 2018. They knew it's going to happen so started thinking long term. They had to constantly work on systems to scale.

They used to hit peak around ipl final and it used to stay there for sometime.
And in the 2019 ipl, they went from 48k to 400k tps. This is the problem they've to solve at Dream11. A huge game may come in like MI vs CSK, so much interest comes into the game and everyone want to play, be active at the system in the same time.
Majority of the requests are read. They hit a 2.7M user concurrency on the platform. They've 40M rps on the edge services. With read traffic, they can do all types of replications and sharding but they also do transactions.
The difference between requests per second and transactions per second is that a transaction might consist of several requests. In JMeter a transaction is - pretty obviously - represented by a Transaction Controller.

There are lot of transactions involved. People are trying to create and update a team and people are trying to join the contest which is closer to the match deadline. When they want to join the contest, they need to add the money at their balance. They are trying to deposit the money so that's a transaction.

Scale:

400 contests joining per second at peak and 24k deposits per s.

When the tosses done, people will rush to their system and edit the teams.

So there will be 15k team edits per second.

The largest contest they served is more than 40L people for a single contest.

They've created a Guinness world record for largest no. of contest joins in a single match. It was more than a crore. They achieved all this with a tech team size of less than a 100 people.

The distributed system at the backend and the scale that is actually provided was built by a team of 25 backend engineers.

The interesting thing is traffic is not linear, when toss happens people come into the system and that will be the first time one can see other

people's team. During the game interesting things can happen.

Lets say someone picked Virat Kohli or MS Dhoni as the captain and they start playing so well. Then user will get a lot of points. Lot of people selects them and when they come to play, everyone will come to see the points. Because of that these external events has a direct correlation to the systems.

Within less than a minute they go from 2-3 million to 10M traffic.

Thats why one of the best features by AWS, auto scaling doesn't work for Dream11.

Because by the time Amazon realises that Dream11 needs more capacity users will come and gone.

So Dream11 to provision for peak and build systems to ensure that even if there is a surge they can handle at affordable cost

How did they manage this? Let's quickly walk through the journey they had.

Everything starts with a Monolithic approach.

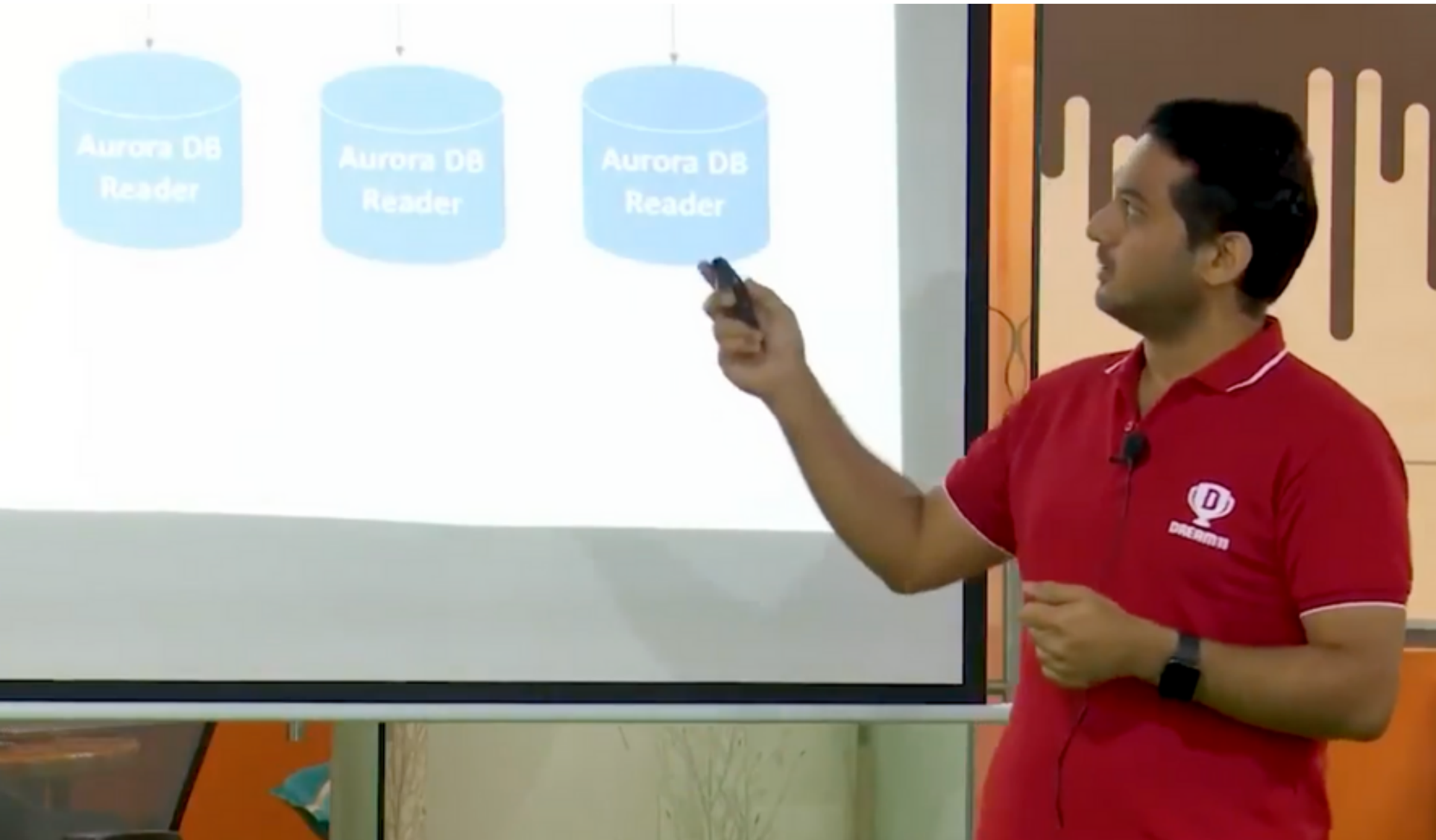
We need to care about how product market fit and what the user is building.

In 2013 - 2016,

The first attempt they did was to migrate to Amazon Aurora as they wanted to buy and solve their issue as any existing solution wasn't present

Amazon Aurora is a RDS that amazon provides and it works slightly differently to MySQL set-up.

Some of the characteristic's of Aurora gave them a bootstrap or jumpstart to horizontal scalability.



In majority of MySQL if we see the slave databases, when the throughput in the master increases , the replica lag starts growing up.

One beautiful thing about Amazon Aurora is they guarantee that the replica lag will not be more than 20ms.

They've the capability of adding 1 to 15 slaves (read replicas). This will enable to put all the write traffic to master and all the read traffic to slaves. This helped them to get read after write consistency. There were some cases where even 20ms was too much for them.

For 99% of the cases this worked, so they placed their DB with Aurora for horizontal scaling.

They've an abstraction built in front of read endpoint, they don't care of what's happening behind.

Users just need to specify readers endpoint and that'll do the load balancing.

Their dependency on devOps reduced, if master gets overloaded another master gets spins up instantly.

Before the start of ipl 2017, they expected a significant traffic. They had a team of 8 backend engineers to handle the traffic.

They prioritised the things that can hit the hardest.

They identified the following few points:

1. Everyone coming to the Dream11 will go load home page and go to the match that's most popular and check the points. That's api will be hit the most. They went upto a million contents in a single game. Homepage consists of the matches and once we go the single match, we've contests. Every time the number of slots left for the contest is changing dynamically at the UI. We can cache some part but sone part is very dynamic in nature.

2. For fetch contests, they ran multiple crone's every 5 seconds and all of them went to the DB and populated the cache. And then they started serving the data from cache layer to everyone.

3. For leader board, they made use of mysql query earlier which calculated the board for all the contests in a few seconds but they realised that this probably might not scale. That's why they went to the data engineering NoSQL DB and introduced Spark.

4. Ranks are calculated every minute for every contest. And at backend also they had a desktop with 4 different users so that was a very complicated set-up.

5. All the contest joins happened in the MySQL DB, the Aurora DB, once all that happened and the match deadline hit, they used a data pipeline to translate into s3 and they used to read data from s3 using a spark cluster.

And then used to do calculations, create initial multiple views at Cassandra and every minute spark job is to run and constantly update the stream data. This was their first experience in creating a data engineering solution.

They slowly started the transition into this system while the primary MySQL is running.

At the first day of IPL, the legacy solution started working. The Query is taking eight minutes to run.

With the MySQL , if the data increased crazy things can occur and had to make the data streaming solution work. And they were able to scale successfully. They served the entire IPL using this solution.

Every IPL they introduce a new system and get a lot of learnings. They work on the learnings and prepare for the next IPL.

For capacity they did the work, and they were sitting at the IPL nervous at make that function. As soon as the traffic hit they had an outage.
And they identified that they are short of servers. They underestimated the amount of servers they would need, hence **capacity planning is very important.**

They started to blindly throw servers in every possible way and it worked.

They started to blindly throw servers in every possible way and it worked.

They were fine for a couple of weeks but towards the final, they again had an outage and started to debug. But they couldn't figure out what actually happened. They again increased number of servers then after 2 days , another outage happened. Then they took this seriously.

Then they found at AWS Cloud Watch, that for ElasticCache.

N/W out bytes kept increasing linearly and then became straight or constant.

So they were hitting the limit of network out bytes. They reached out to Amazon and they were confirmed. So again they scaled their instances.

Their monolith Aurora was working fine at scale, but they were running so much aggregate queries on them. They worked with Amazon for two years to solve this.

The feasible memory at the master kept on dropping and vanished. Ideally it should be constant.

Aurora panicked, slave became master and fail over occurred.

If fail over happens during the game, it can disrupt the user experience.

So they had to remove aggregate queries from main transactional DB.

The third outage they had after wrong capacity planning and aurora outage was regarding **DNS lookup**. They started getting so much scale.

DNS look up started chokking during the scale. They introduced a NSCD cache on all the EC2 servers and this isn't what they expected. It represents their low grade with load and performance testing that time.

They were happy with the overall IPL even if their were few outages and 10x traffic was recorded.

They knew that monolith won't survive in next IPL. Migration from monolithic to micro services was difficult for them. They had to do a lot of POC.

The data model is very much tightly coupled. And because of the nature of the contest people were joining them and paying money.
They have to give a very strong consistent experience.

According to the limitations of CAP Theorem, eventual consistency is not a very good option for them.
The CAP theorem stands for Consistency, Availability, and Partitions and expresses that a distributed system generally produces two of three properties simultaneously and characterizes the compromise between consistency and availability when a partition exists.

Eventually they realised that they've to loose their consistency a little bit. But their goal was always to be as consistent as possible.

With monoltihic approach, transactions break. Money is getting deducted, promotions are made via a single transaction.

The moment they split to micro services , the transactions are happening across.

Contest Micro-service they started working on.

They had a monolith , all the services were talking to it and suddenly things needs to be changed.

They made use of Orchestration a.k.a Edge layer to identify transitions behind micro-services.

They had to build a business logic for routing before starting to breaking things.

They created an edge layer and called it coordinator. Everything goes to the coordinator and it decides what needs to be done next.

The design started constantly changing, they enabled automated testing and deployment.

They've to improve conversions all the time.

They started with UI Testing , Automation first and that was a mistake.

Backend testing can give a breathing space to QA to keep moving forward.

At Spotify Squad Model, there are squads and they put cross functional teams are put together and they give them a focus area.

They're inspired from that and they created a Dream teams which is a combination of UI/UX, PM, Backend , Frontend and they provided them the areas of focus so that they can operate and that was a de-centralized team.

They wanted bottom up culture and not a top down decision making. And they allowed the team to decide what to do.

They brought on scrum, didn't do that well, moved to kanban board and moved to scrum back again.

They realised that they need to start investing in data because its going to be their competitive advantages as they go forward.

It can be useful for understanding the behaviour and given them a good user experience.

So they started forming analytics and data science team. That identifed their direction.

So they started forming analytics and data science team. That identified their direction.

For the next ipl,

1. Contest Service Was Done.
2. Coordinator was ready
3. Routing Happened
4. They started moving some matches to the new service.
5. For the leaderboard they were confident that Cassandra and spark would scale.
6. They isolated some of their micro-services for better readability.
7. For authentication monolith service, they created a different data pipeline to a different database and placed application servers in front of them where people can authenticate
8. For team , it was completely isolated. Every match has a joint team, it has nothing to do with the historical data.
9. They created a very clean micro-service with different strategy and fresh database.
10. Tour is like a metadata for them, so every single call needs to identify what match, what's the time, who are playing and what are the squads. All this data is very very important.
11. They went back to the cron crashing strategy, they passed multiple crones. They created a DB model from redis. So Redis became a backbone for their tour service.
12. Each service they need to decide how perfectively they can carve it out. They can do reverse data pipeline if required.

13. From the previous experience they had, they got to understand that load testing and capacity planning is very much important.
14. They spent a lot of time to understand the traffic pattern from last year
15. As they were able to mimic the production better than earlier, they were better prepared with alerting and monitoring.
16. Because with more number of services, they don't know whats happening when they slows down and monitoring becomes crucial.
17. It's important to invest in dashboards thats helps in building a very fast tracing.
18. With every scale, company starts getting different challenges.
19. They hit some scale challenges at AWS instead of themselves alone.
20. ELB was scaling as they were growing up but it had a capacity.

Key Learnings:

1. ELB pre-warming
2. Socket limit
3. Amazon Cloud Front
4. Redis Hot Hey
5. Bugs

Their traffic was surging so much that Amazon ELBs were not able to expand. So they had to work with them and pre warm the ELBs for the traffic. They informed AWS what kind of numbers Dream11 expects and they were prepared for them.

Load test can never 100 percent simulate the production traffic.

They used jmeter to hit 100 servers. Actual Behaviour is bit different.

At local it was working fine, they were able to handle rpm but at production they started getting issues in edge layer.

CPU and memory was fine but because of number os synchronous requests from client, they were running out of sockets.

They needed c4-2x and c4-4x. They out-scaled the servers and doubled the number.

They had CPUs and memory but didn't had required number of connections so they configured nginx settings, load limit settings and did lot of operational settings.

They had a beautiful dashboard for cloudfront when sitting in a wardrobe.

About 10 people from 100 people they day in office that time used to come and report bugs.

This was not a good sign and they found that the entry point was cloudfront.

They were only monitoring the traffic that reached the edge layer but their was 10% of the traffic that was getting 5xx error from the cloudfront mirror itself. Their was some cache put in at the amazon layer and they were not able to handle Dream11 scale. Dream11 had a good relation with AWS. They reached out to the cloud front team.

They helped them and started monitoring 5xx errors from cloud front as well. Earlier they used it for alerting and visualizations.

Even if 1% error happens in cloud front they get a alert. Caches and Redis was configured beautifully, in elastic cache they've
one single cluster. They've sharding but limit is 5 for no. of replicas that time.
Sharding and partitioning are both about breaking up a large data set into smaller subsets. The difference is that sharding implies the data is spread across multiple computers while partitioning does not. Partitioning is about grouping subsets of data within a single database instance.

Some keys were so hot that they were maxing out the capacity of all the screens itself.

There are 2 options:

1. One can cache the key locally into the system somehow.
2. We can create multiple versions of the key and rotate across shards
3. Their will be a hot key problem in the system as it scales up.
4. Problem with moving fast can result in critical bugs

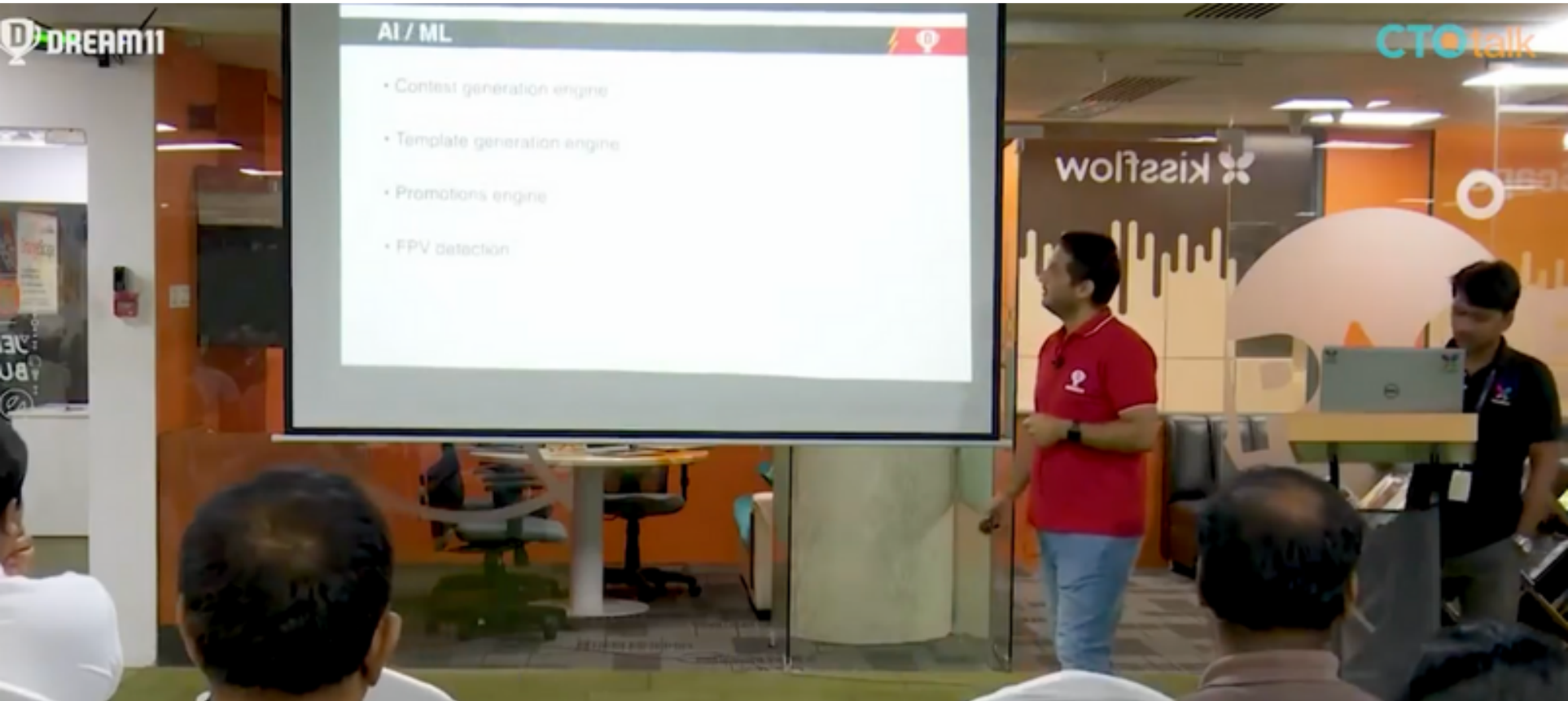
HotKeys usually have long CPU run time, which deteriorates Redis performance and affects other requests. Hotspot on some Redis nodes/machines, instead of on keys that sharding to different Redis nodes, often prevent you from taking full advantage of Redis Cluster.

As they were moving international, their interest in the game will keep moving. So they had to start investing in systems that actually are almost infinitely scalable.

They spent 2 or 3 months, on the R&D for their architecture. They had to add featured to the game because of more users engagement. At the same time they had to built platform components.

They had to build a centralized horizontal layer and they started looking at the DBs they didn't had earlier like inmemory Volt DB that does things diff. from traditional MySQL DB.

They introduced Aerospike and they migrated tour service to aerospike. Their aim was to completely have microservice driven architecture before next ipl and not monolith based



Data team was growing constantly.
They had to reply on ML to understand what kind of contests are created historically.

They created a contest generation engine for that. At this point , they generate a millions of contests per a single day.

So they made 100% of contest generation completely automatic based on template.

They also had to configure out what kind of contest needs to be created.

May be they need to tweak the contest structure itself.

May be they need to create a template generating engine and figure it out what's working and what cannot be working.

They invested on identifying what promotions should be given to what user and at what point of time.

People can try taking advantages of promotions, so they introduced Fair Play Violation (FPV) Detection System. They made lots of checks and rules involved.

6 weeks before IPL 2019, things were much comfortable and suddenly business took a massive decision to modify the match deadline from 1hr before the match to just before the match

From a backend team perspective changes will be present drastically. People can join at any time and create teams. They'll wait till the last minute to get most information related to the game to make decisions for their respective teams.

Scale was never an issue in the team service . Max scale which they served at teamservice before was 300 tps. And in worst case scenaio, they tested against 50k tps. This was crazy and following pic depicts some the design patterns to solve some of the problems they had.

One of the easiest thing is sharding. For team service, anything that is user centric where it doesn't have to do anything with each other can be

sharded.

They're like dividing the database into various segments and they did the sharding.

Their leader board is re-calculated every minute, but for more than a few million contests across a millions of users.

It was very important for them to show a consistent view.

They started flipping the score atomically once the leaderboard is generated.

They started putting a event entry for every leaderboard calculation they had.

They actually wanted to put a local cache layer. Every time a user is generated, the local leaderboard is cached.

One of their contest is mega contest where majority of users join. The first set of data doesn't change between two iterations.

They had the ability to cache it locally on the servers. But to knew that the data is changed they used the redis pub-sub where the publishes the message to redis and everyone knows its time to actually update the event.

Memoization :

Their tour service had a metadata information about a lot of contextual information about calls.

As they scale more and more features the scale on two services increases. They are multiple ways to solve this, when the metadata changes they can push the metadata to all the services.

But in their case, with the service like tour. There is a time (round lock time) where the match closes and people goes.

They cannot afford data not getting propagated to one particular system showing a wrong information to the user.

They saw that tour service started hitting 30M, 40M, and 50M rpm.

Even if they horizontally scale, it's very inefficient. They introduced memoization and they realised tour information across user requests is how much. It lets users to group a bunch of requests together.

It sends one request ahead to the end system. It gets the data and all the get the information will be divided.

So once they introduced this to the system, throughput of metadata drop by 75%. It made a significant change into the system.

This is what they played with Volt and it worked significantly. They started introducing into a lot of servers. Their edge layer with all the traffic comes in, they group it.

OC Read Write Segregation:

One of they actually did is the introduction of Volt DB which is the in-memory DB , it's great DB for writes.

If they had read/write at the same time, it was creating a lot of issues for them.

They were preparing a system for 10K joins per second and one of the challenges in contest joins is there's a counter that maintains how many people had join the contest.

In typical MySQL world, what does it means is if we won't make a joint entry and update the count. It doesn't get serialized within that block.

They made the throughput limit at MySQL and it couldn't scale.

They introduced Volt DB for that and it allows to re-write to MySQL by throwing the event into kafka. They're actually populating MySQL and some of the use cases. Reads and writes were happening separately.

Their systems needs to be consistent and lots of quality checks were put into this.

One of the things they did is to hit at one place and do a write as well.

If they do a dual hit, the pipeline event that comes in may be with a combination these fields might have some guarantees.

They put some interesting patterns , they put a write and in a dual write they put a flag there.

Actually event derived the actual flag.

At the same time they ran scripts on batch processing combining DB from multiple services and try to have various layers of QC (Quality checks) to identify which data is missing.

It wasn't an easy problem to solve, they did solved for this but moving forward they invested heavily . A lot of instances were falling into this.

They invested on what kind of quality checks on this.

Saga is a newest entry.

With the problem of distributed contest join, if there's a request for contest join and then they go to wallet service for the deduction of the money and then they come back for the contest join.

Using saga's they built a central framework for such things.

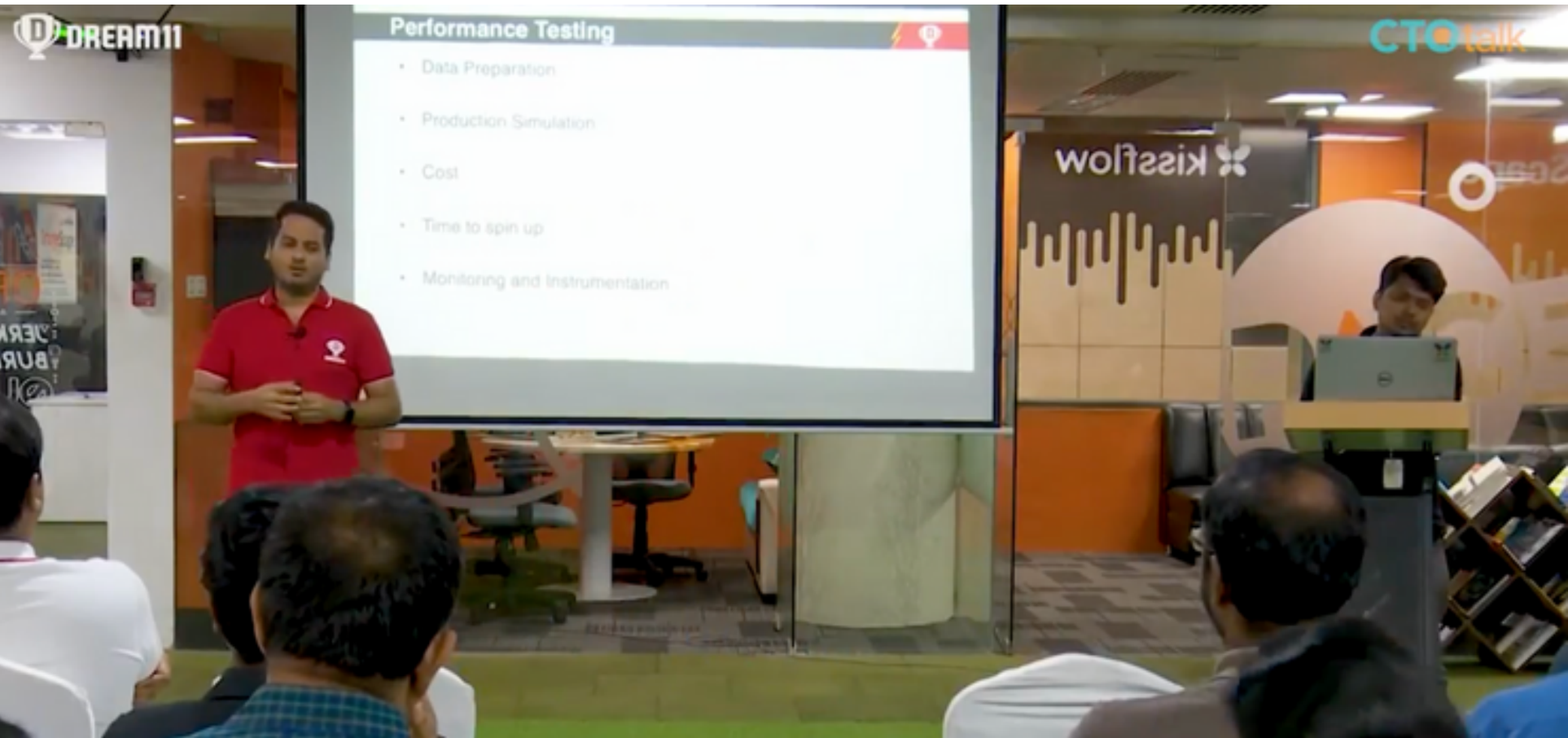
What happens is all the local transactions are grouped together and the centralized system actually does one at a time.

And if anything fails in the chain, it knows how to repeat or go back and revert the original transactions or try it again.

At the developers point of view if specifications are ready, they can do the required transactions.

And one central system will constantly keep doing the things for them. This is the design pattern they introduced.

A point is that once they start hitting the scale, they constantly be faced with though challenges. And they do out of the box thinking.



Performance testing is great and they got a lot of information related to it.

They could predict, simulate and do a lot more. One of the biggest challenges which comes to the performance testing is the data and how do they perform with more than 50 micro-services.

They should generate seed data that lies across all these services.

Generating the data is a huge count and they can never simulate the production equally.

Major of their tests are running fine.

Production costs are highest in any environment. And they are preparing for 4x of the traffic they get in production.

Load environment needs to be scaled lot more than production and different people are running different load tests at different times.

They completely use spot instances in load environment to get the cost down. And its very important to build automation that scales up, down and fast.

Lets say it takes 6 hours to set up INFRA and do the test. Automation and scaling is very important related to auto scale.

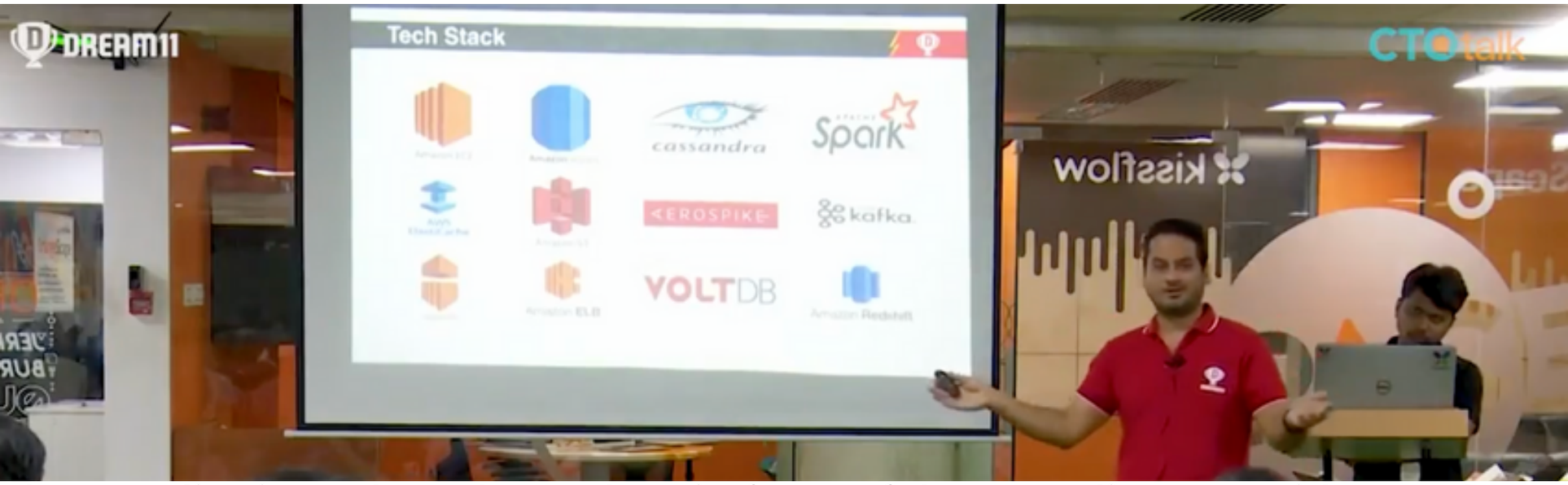
Once they do the load test, they don't get the desired result.

Monitoring and instrumentation are crucial to exactly tell where the bottlenecks are.

We're constantly breaking to identify where the meter is running. We've to make sure that we're constantly working on that.

Basic Architecture Of Dream 11

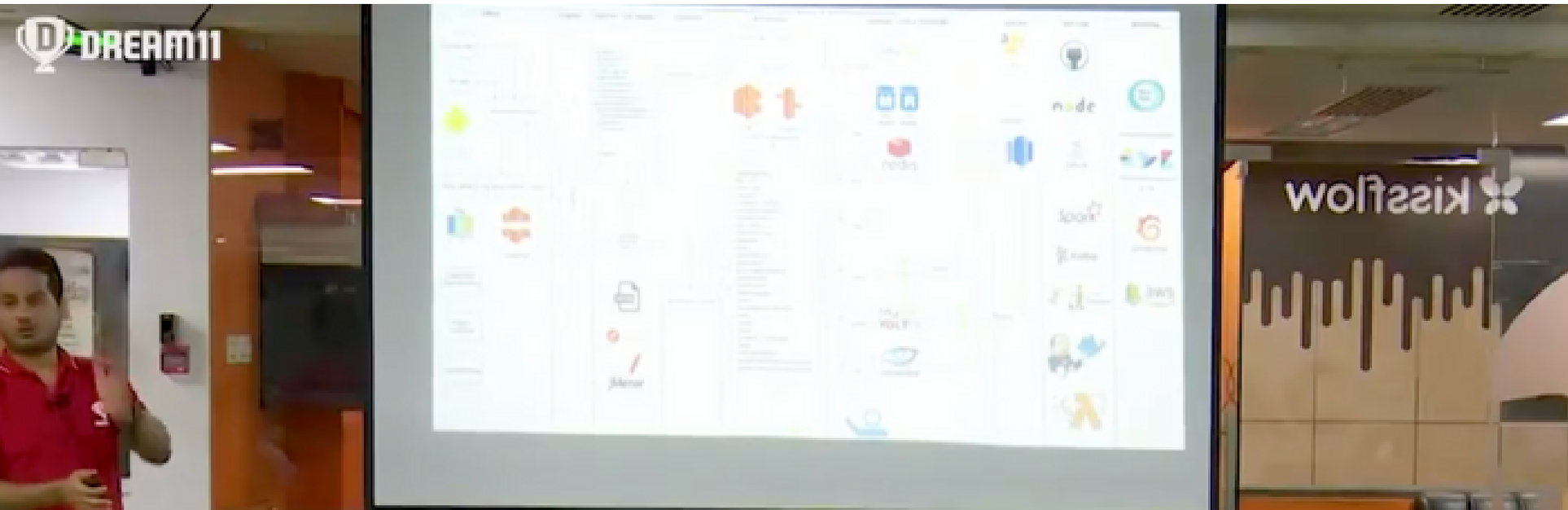
Technology Stack Of Dream11



They earlier introduced GraphQL in front of the coordinator. With GraphQL schemas can be used for getting the data.

Clients just ask for what they need. Component architecture becomes lot more easier with it. And it gives lot of power and flexibility.

Backward compatibility becomes lot more easier when it comes to service oriented architecture. They've RedShift as the warehouse.



They earlier made use of Google analytics and with their concurrency of 1 or 2M, the page doesn't load. It was expensive. They might need only 5 params but they must be extremely fast.

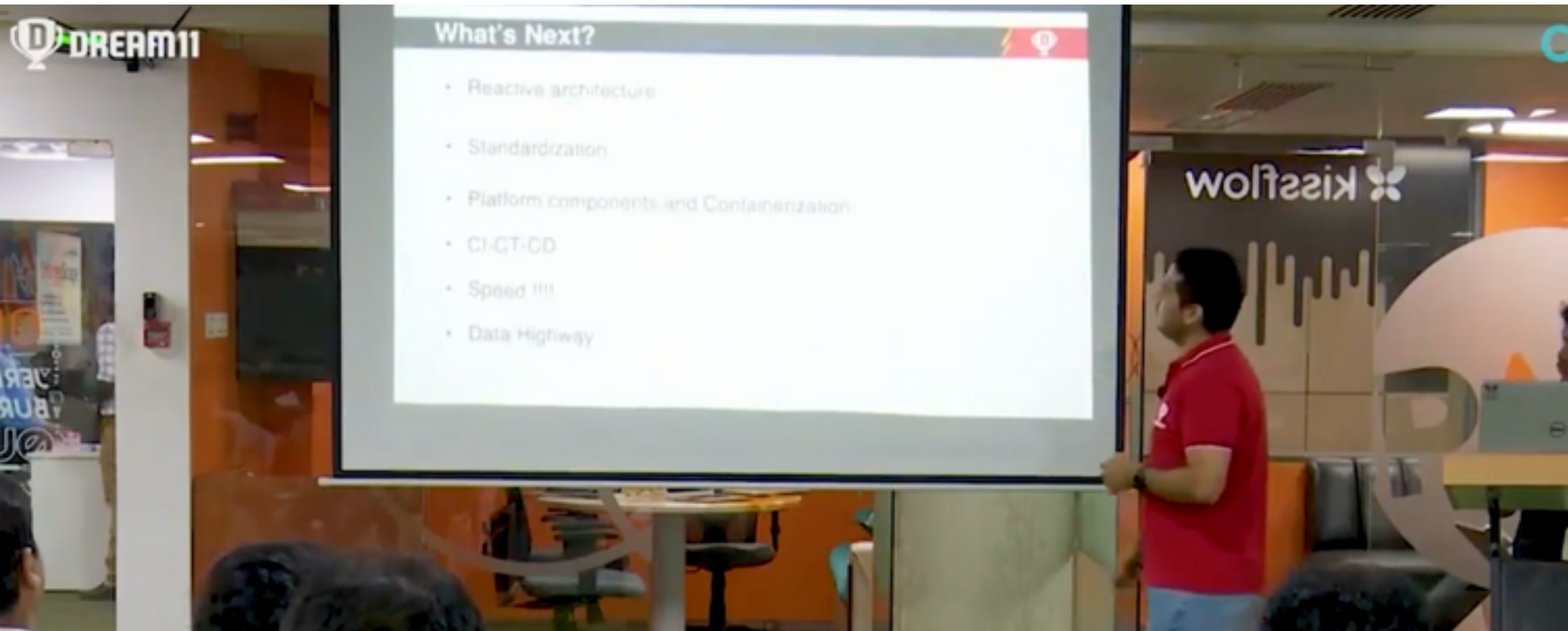
They started with Node.js , moved to java and now they are working with Scala.

They used to make use of loggly which is a managed service for ELK and they used to lie on loggly for alerting efforts.

During ipl, loggly ELK Stack lagged by 8 hours. This couldn't scale for them.

Now their ELK handles alerting in real time.

Advantages of taking it to the cloud to optimise that for the exact use case.



They were moving their favourite contest service into AKA. Standardization is the key. Frameworks also helps to avoid redoing the same things.

Once they move to the continuous testing they need to move to deployment.

They want to move faster even with the scale they're currently having.

Data highway is the code name for their analytics project.

FanCode is their next app they created which is unlike a typical fantasy sports system for better customer base. It's more of a content app.

They were building a republic of sports.

zero percent of adds they offer along with 100 percent of sports. They started live streaming.

FanCode also consists of cricbuzz today. They enable users information and do research at fancode so that they can create best available fantasy teams on Dream11.

Here's the link for **Dream11 Tech Blog**:

https://blog.dream11engineering.com/

Tech stack for android and iOS : native. For fanCode they've react native.

For web they built a PWA and there's no framework.

For high scalability , they do capacity planning linearly or exponentially.

Security advise while dealing with credit cards and customer's money:

They don't store cards in the system but they pass through it. They ensure that one of the api's are exposed.

Their development team don't have access to sensitive data and few environments.

For restricted users they've a jumpserver with the ssh and password

For s3 they'll only give access to specific sqs queues and provides read access to various aws resources but not write access

Following are the challenges they faced while rolling out to production using kubernetes:

1. Their billing system should be 24*7 up
2. It was hard for them to debug

With a smaller company, one can experiment things. But once they become a bigger company and mission critical stuff running, the process of adapting something new and taking it live becomes longer.
They gave an example that N requests will be plugged together, it will be carried forward to the packet service and returned to end users from there.

If 1000 requests are coming, reqs coming to the edge layer are synchronous.

They had the challenge to make sure data across servers is consistent.
They had the downstream services to send the expiration time.
Cache invalidation should happen at the exact same moment. They wanted downstream services to control how long it can be memoized.

If there's a round trip of 5secs, all the n reqs can be blocked for ms. They sever 45M rpm but their avg. rate is 645ms

Their throughput is so high that even if they cache for ms, it will drastically reduce the load. They did play around with some sort of req. tracing. If it's not expensive, they'll built by own.
They also worked with bug bounty and hacker rank for security.
With cassandra,they're using computer rathen than storage optimised sol as the performance was better.