Project Experience 1:

Client's application is an Online Cab booking service currently operating in more than 10 cities and this is a very high volume application consisting of 16 modules, Relational databases with multiple Read Replicas & Non-Relational Databases, Caches, and Data Analytics platforms. To give you an idea of the size of this application, usually, we would be receiving around 15000 requests per minute during our regular working hours.

Architecture:

All the modules are Dockerized, and hosted on AWS's EKS through Terraform templates considering all the best practices for HA, Security, Scalability, and cost Optimisation to meet the business needs. Our CI/CD setup consists of CodePipeline which will fetch our Code base from Bitbucket, after manual approval, the docker containers will be built using AWS CodeBuild, and Deployed in a staging area where we will run our test cases and make sure everything is working as expected. Once it is done and everything runs successfully, we will deploy the changes in production using AWS CodeDeploy.

One of the unique challenges with this application was to handle peak traffic while we sent out notifications. When we do this, our incoming requests multiply by 4. To handle this, we built an autoscaling logic that will scale our servers beforehand using a custom lambda function.

We work closely with the Developers and Stakeholders (non-technical people) to address their needs and communicate the risks and implications of their choices. We have built a few custom CloudWatch Dashboards to view all the application metrics in one place. Developers will have a set of dashboards and stakeholders will have a few dashboards where they can view the metrics as per their need.

We use NewRelic, and ELK Stack for monitoring our application's performance. PagerDuty & SquadCast for alerting & incident response.

Recently, our client wanted to integrate with a 3rd party provider for their Data Analytics solution. The 3rd party provider wanted our client's data in a S3 bucket with data in CSV format. To explain it in detail, we had the analytics data (data that needs to be shared with the 3rd party provider) in AWS DynamoDB and we needed to build an ETL solution to transform this data in a format that the 3rd party provider requested and push the transformed data in the S3 bucket. We explored a lot of options and finalized 2 solutions, one is to use AWS Glue and another one is to use DynamoDB streams with a custom Python lambda function for the ETL. Though AWS Glue is a managed service specifically built for

these kinds of ETL works, it was costly compared to our Custom lambda function approach. So we conveyed both options to our client, and he agreed to go with the cost-effective option.

So the final solution that we provided is as follows:

Since the data to be transformed is already in DynamoDB, we used DynamoDB streams that triggered our Custom ETL lambda function every 5 minutes or if the stream size reached 10000 records. This Lambda function will transform the data & push it to the S3 bucket & through S3 bucket policies, the 3rd party provided will fetch the data in CSV format to do the analytics part. This Solution worked, and the client was very much satisfied with our output.

There was another need where we needed to build a data analytics pipeline to send our analytics data in S3 to Microsoft Power BI. There were a lot of intricacies while doing this, since CSV format is costly for querying we pushed the data to S3 in parquet format. Since there isn't any straightforward approach to sending our data in S3 to Microsoft Power BI, we configured a Data gateway with Athena connector in our VPC that will fetch data from our S3 bucket and push it to PowerBI.

Project Experience 2:

We recently undertook a client project that entailed the migration of five substantial monolithic applications and a cluster of microservices, all hosted on AWS Elastic Beanstalk. Our team was tasked with migrating these workloads, spanning testing, staging, and production environments, from AWS to GCP's Kubernetes Engine. It was of utmost importance to adhere to the best practices concerning high availability, security, scalability, and cost optimization to align with the client's business objectives.

The challenges we encountered during this process were as follows:

Challenge 1: The initial monolithic nature of the application required us to reconfigure it into multiple microservices before migration. To achieve this, we dockerized all the modules, deployed them within GKE (Google Kubernetes Engine), and rigorously tested them for accuracy. This culminated in the development of a comprehensive migration plan for the production environment. Subsequently, we applied the same plan to migrate our staging workloads to GCP, ensuring its robustness and fault tolerance. Upon finalizing these steps, we seamlessly migrated our production environments to GCP.

Challenge 2: The migration of the database (PostgreSQL) presented another critical area where we meticulously devised a Master-Master replication setup between our AWS RDS

and GCP's Cloud SQL. This configuration was essential to maintain data integrity and avoid any application downtime.

Challenge 3: Despite our extensive experience with AWS services, the DevOps Engineer leading this migration had to acquire a new skill set from scratch when it came to GCP.

The results of this endeavor were remarkable:

1. All applications and databases were successfully migrated to GCP without significant interruptions.
2. We accomplished these tasks within a tight two-month timeframe, adhering to best practices for high availability, security, scalability, and cost optimization.
3. Our strategic planning and efficiency efforts led to a reduction in cloud infrastructure costs for our client.
4. The client expressed high satisfaction with the outcome.