Hello, and a very good evening.

Let me introduce myself. My name is Amit Kumar Verma, and I have been working with **Brindavan Bottlers Pvt. Ltd.** for the last 1.4 years. I have a total of 7 years of experience, out of which for the last 3 years, I have been working as a DevOps engineer.

Let me first explain the use case. I am currently engaged with a US-based customer. The use case is that the customer has turbine engines installed, and if the temperature of the turbine engines goes beyond a certain threshold, there is a huge breakdown in the system resulting in a significant loss for the customer. To handle this, there is an application team that has implemented a monitoring solution. These applications are running in both monolithic and microservices architectures. For monolithic and microservices, the frontend is built with React, while the backend is a mix of Python, Java, and .NET technologies. The database is hosted both on cloud and on-premises. This is the overall architecture for the application.

Suppose any vendor wants this solution, there is a separate application team consisting of 10 people, and we are supporting them from the DevOps side. I am acting as both an architect and an implementer here. Talking about my roles and responsibilities, once we develop the high-level diagram and get it approved by both the Microsoft architect and the customer architect, then myself along with my team implements those high-level designs by creating the low-level designs (LLDs).

For implementing DevOps best practices and solutions, we have selected Infrastructure as Code (IaC) using Terraform. In Terraform, we have written generic modules considering dynamic blocks for each map of objects and optional attributes. All optional attributes available on the registry have been integrated with dynamic blocks and for-each iteration. So, if any user wants to add optional attributes in the future, they don’t need to modify the module code — it’s already generalized. We have created these generic modules starting from Management Groups, Subscriptions, their Associations, to Resource Groups (RGs), VNets, Subnets, including Peering, VPN Gateways, Load Balancers (both internal and external with Public IP), Application Gateways for multi-site deployment, Azure Front Door for global routing, and Azure Traffic Manager. These services all have generic modules.

Apart from this, we have also created modules for NSGs, Firewalls, UDRs (routes), and monitoring solutions, backup solutions, DR solutions — everything is modularized. Each module is stored in separate repositories using a multi-repository concept with proper versioning, managed within Azure Repositories.

From here, we provide solutions across three environments for the customer: Sandbox, Development (or Pre-Prod), and Production. In the Sandbox environment, we provision minimal resources like a test subscription by calling the Management Group module, Subscription module, associating them properly, and putting the right dependencies through pipelines. We are deploying this setup with state files stored remotely in Azure Blob Storage with Customer Managed Key (CMK) encryption enabled.

For Development, we call all required modules as per the customer’s requirements — Management Group, Subscription, Associations, RG, VNet, Subnets, Peering, etc. — by creating a parent module with proper dependencies. We create generic Terraform configuration files (TFRs) and maintain code using trunk-based branching strategy in Azure Repositories. The pipeline setup is done through multi-stage YAML pipelines using Azure DevOps.

When any new VM needs to be added, we first clone the existing repository (which has the parent job), create a feature branch, update the TFRs to add the new VM, push the code to the feature branch, and raise a PR. As soon as the PR is raised, the pipeline runs automatically. In this pipeline, the first stage is the sanity check where linting is done using TF Lint, security scanning is done using Checkov, vulnerabilities are checked with TFSEC, and sensitive data is scanned using TruffleHog. Additionally, we have integrated Chef InSpec for automated test case validation.

If all these checks pass successfully, the plan stage gets triggered and then manual validation (GenTest Job) is required. Once the manual validation is approved, the changes get deployed to the Development environment and the new VM gets added to the inventory. The same flow applies when moving changes to Production.

This is how we handle infrastructure provisioning end-to-end, whether it’s a monolithic application or a microservices application.

When it comes to microservices, we also manage Azure Kubernetes Service (AKS), Azure Container Registry (ACR), Azure Key Vault, and Bastion Host setups. Generic modules for these services are written and stored in a multi-repository structure with versioning. Landing zones for AKS clusters are created through Azure DevOps pipelines following DevSecOps practices and a trunk-based branching strategy.

Once infrastructure is ready, we proceed with application deployments. For monolithic applications, middleware installation is required. Based on the application type, middleware such as IIS for Windows servers, NGINX for Linux servers, Tomcat, Apache, or any other required software is installed. This is handled within the Terraform module itself using custom data scripts that detect whether the VM is Windows or Linux and install the required software.

After middleware installation, application deployment takes place. We use GitFlow branching strategy here. There are feature branches, development branches, release branches, hotfix branches, and a main/production branch. Suppose a new feature needs to be added, the developer clones the repository, creates a feature branch, makes the changes, pushes it, and raises a PR.

Once the PR is raised, pipelines trigger automatically. The quality of code is checked using SonarQube integrated into the pipeline. Vulnerability scanning is done using Checkmarx. Dependencies for Java applications (Maven-based) are pulled not from public repositories but from private repositories like JFrog Artifactory, Nexus, or Azure Artifact feeds.

Once code passes quality and security checks, the build artifact is generated and published to Azure Artifacts. This artifact is then deployed automatically to the Development and Test environments.

After successful testing, the artifact is promoted to the QA branch, where it is manually approved before deploying to QA environment. After QA approval, deployment to Production happens, requiring approval from business stakeholders, unit heads, DevOps, Development, Quality Assurance, and client-side stakeholders. Once approved, the application is deployed to production, and notifications are sent out.

We also have a rollback strategy in place. If something goes wrong, we can trigger a rollback pipeline using the correct artifact version, thereby quickly restoring the previous stable release and minimizing business impact.

Regarding backup strategies: for infrastructure, we rely on Azure Backup; for applications, code is backed up in GitHub, and artifacts are stored and versioned in Azure Artifacts for easy redeployment if needed.

For monitoring, we use Prometheus and Grafana for Kubernetes clusters and Datadog for application-level monitoring and matrix collection. Backups are regularly taken to ensure DR readiness.

We also enforce governance through Azure Policies, both built-in and custom ones aligned with CIS Benchmark standards, to maintain GDPR compliance. Regular compliance reports are generated via Bigly and distributed to the support team for fixing any issues.

Currently, the customer is migrating from monolithic to microservices architecture to optimize cost, quality, and downtime. Initially, they are moving non-critical systems to Kubernetes. We are using Terraform to create Kubernetes landing zones through Azure DevOps pipelines with DevSecOps practices.

Once AKS infrastructure is set up, Docker images are built, scanned with Trivy and SonarQube, and then pushed to ACR. Kubernetes manifests (deployment.yaml, service.yaml, ingress.yaml, configmaps, secrets, PV, PVC) are created to manage deployments. Internal load balancing is managed through Cluster IP services. The Application Gateway Ingress Controller (AGIC) is integrated to manage external traffic. WAF and firewall policies are applied at the Application Gateway level. Ingress rules are defined to route traffic to services inside the cluster.

In the near future, we plan to introduce Helm charts for packaging and deploying Kubernetes resources more efficiently. A basic POC has been completed, where templates are generated at runtime using values.yaml files.

We are also planning to move from traditional CI/CD to GitOps by integrating ArgoCD for managing AKS deployments. Research and development work for ArgoCD is currently underway, and implementation is targeted for the next quarter.

Monitoring continues to be enhanced using Prometheus and Grafana dashboards for cost, security, and performance monitoring.

Overall, DevOps is an ever-evolving journey. In every PI (Program Increment) planning, our focus is on improving existing systems, introducing new automation, and making our processes more efficient, whether it's for infrastructure provisioning, application deployment, monitoring, observability, or governance. Every sprint brings new initiatives toward making our engineering processes smoother and more reliable.

Thank you.