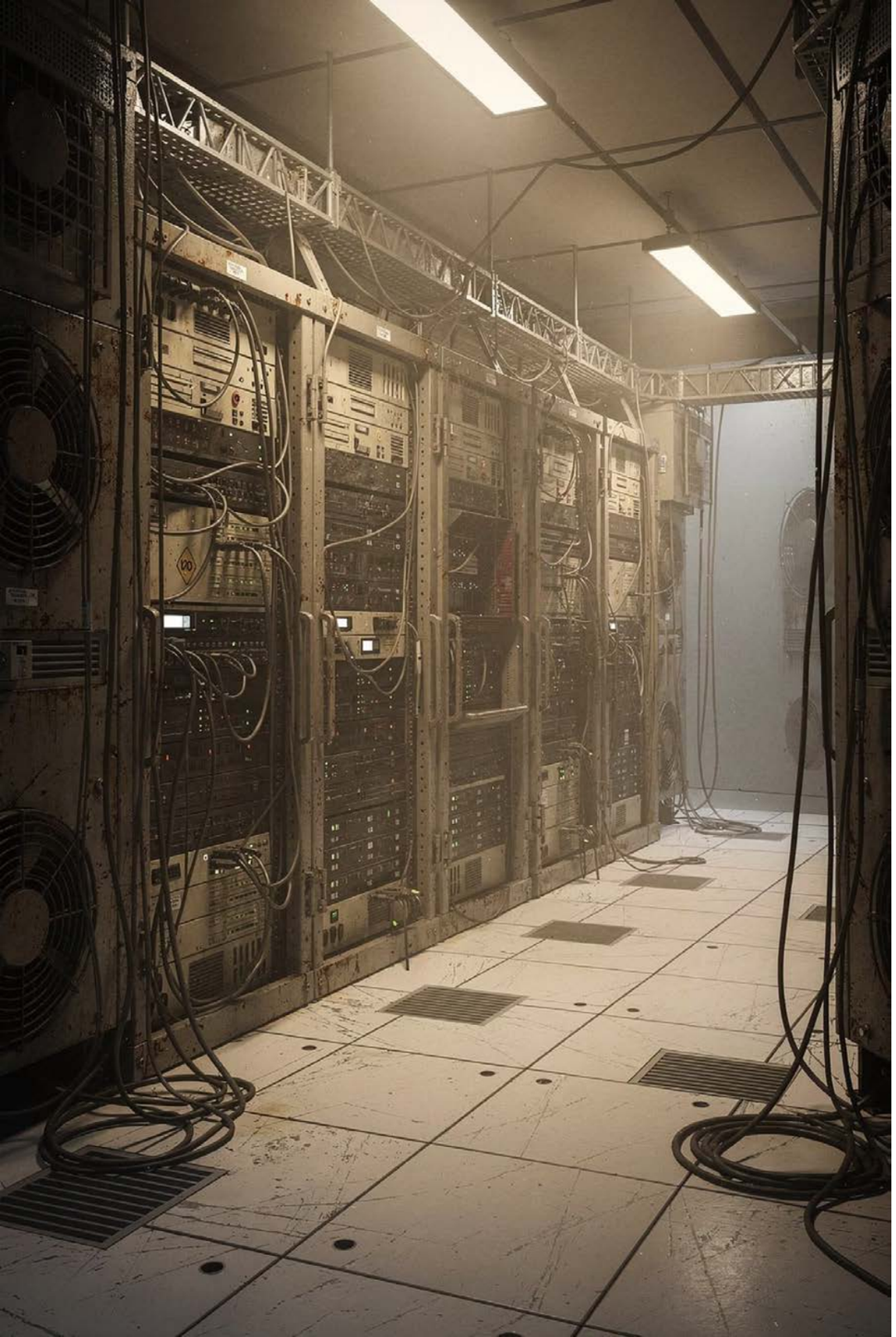
From Lift-and-Shift to Kubernetes: A Two-Phase DevOps Modernization

Presented By : Harris Peter Baskaran
Google Inc., USA

Conf42 DevOps 2026



The Challenge: Legacy at Scale

Where We Started

A tightly coupled monolithic ticket management platform running on VMware infrastructure with Oracle database backend. The system processed substantial daily ticket volumes while maintaining strict service level requirements.

The architecture had evolved over years into a complex web of dependencies, making any change risky and time-consuming.

The Migration Goal

Transform this legacy system into a cloud-native, containerized platform without disrupting operations or violating SLA commitments.

The approach needed to be methodical, measured, and cautious while moving toward modern DevOps practices.

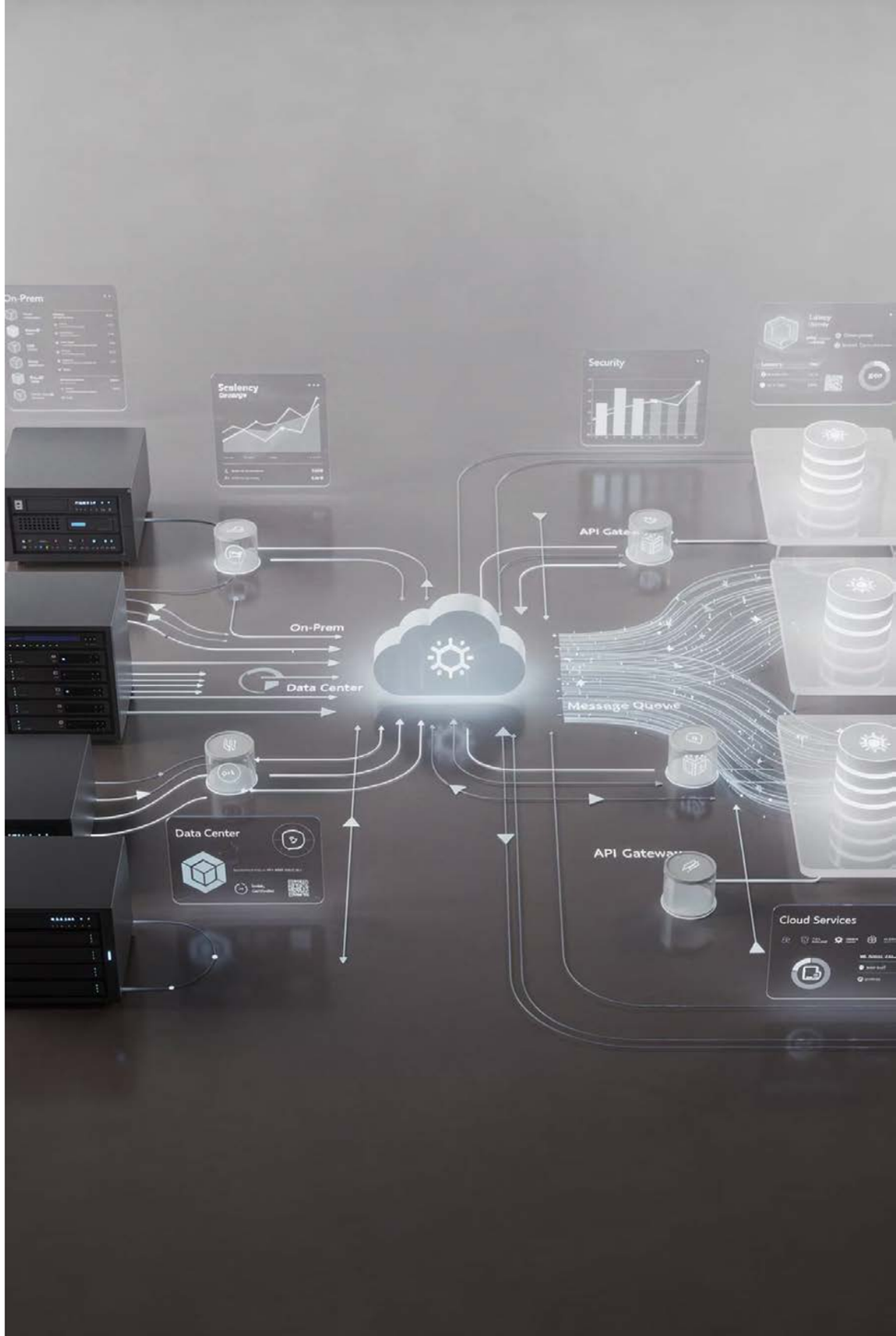
A Two-Phase Strategy

Phase 1: Lift and Shift

Migrate to Google Compute Engine preserving the existing architecture to establish a measured baseline

Phase 2: Cloud-Native

Redesign on Google Kubernetes Engine with containerized deployment and modern data platform



Phase 1: Establishing the Cloud Baseline

01

Dependency Mapping

Comprehensive analysis of application dependencies, integration points, and communication patterns to understand the full system topology

02

Parallel Deployment

Built cloud infrastructure alongside existing systems, enabling thorough testing before cutover

03

Continuous Replication

Maintained data consistency between on-premises and cloud environments during transition

04

Almost Zero-Downtime Cutover

Executed migration without service interruption, preserving SLA commitments throughout

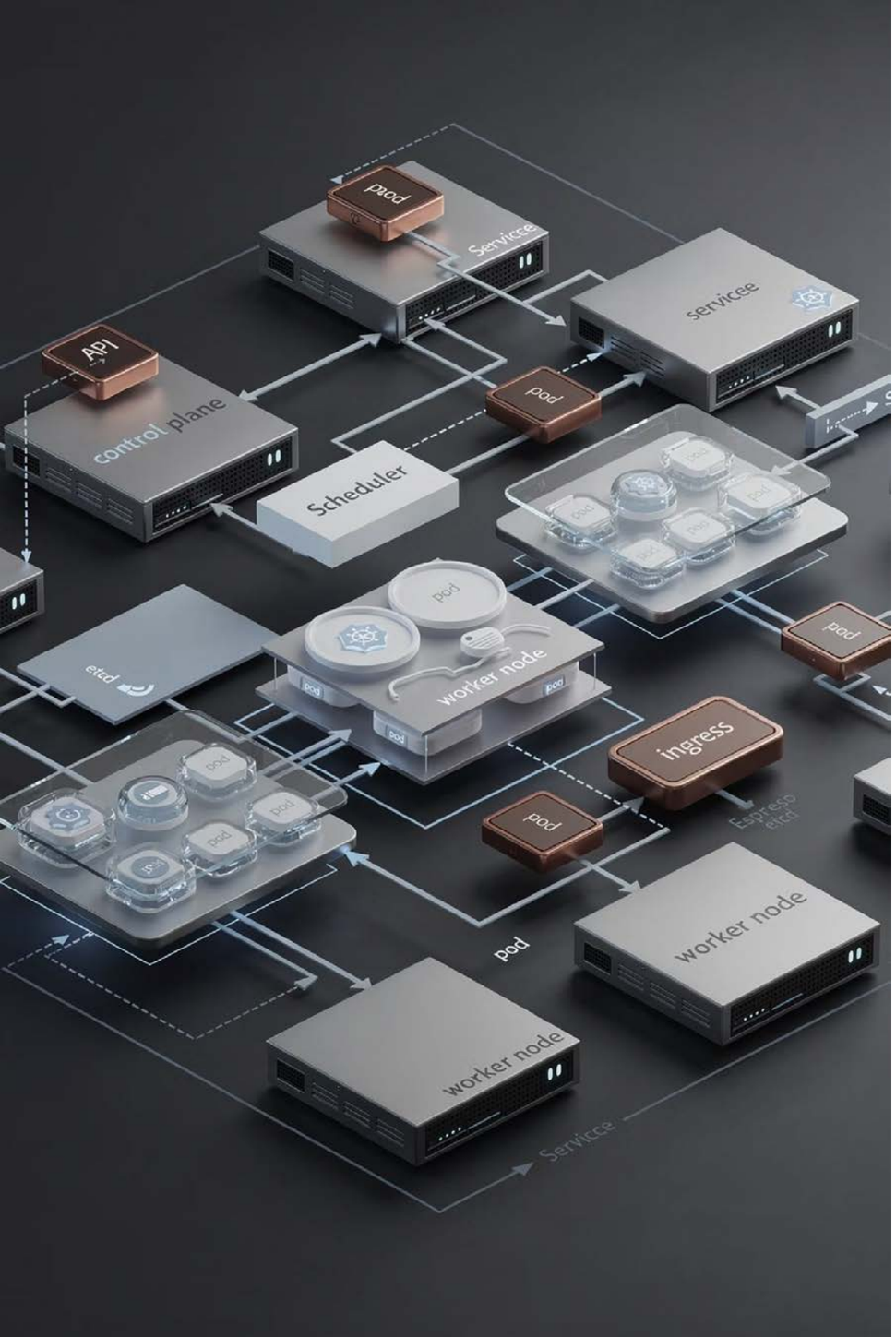
Measuring Success in Phase 1

Establishing Performance Baselines

With the platform running on Google Compute Engine, the team established comprehensive performance baselines across key operational metrics.

- Application latency patterns under various load conditions
- System throughput capabilities and bottlenecks
- Resource utilization profiles for compute, memory, and storage
- Network performance characteristics

These baseline measurements became critical reference points for Phase 2, enabling the team to validate that the containerized architecture maintained or improved upon existing performance.



Phase 2: The Kubernetes Transformation

Container Platform

Migrated to Google Kubernetes Engine for orchestration, scaling, and deployment automation

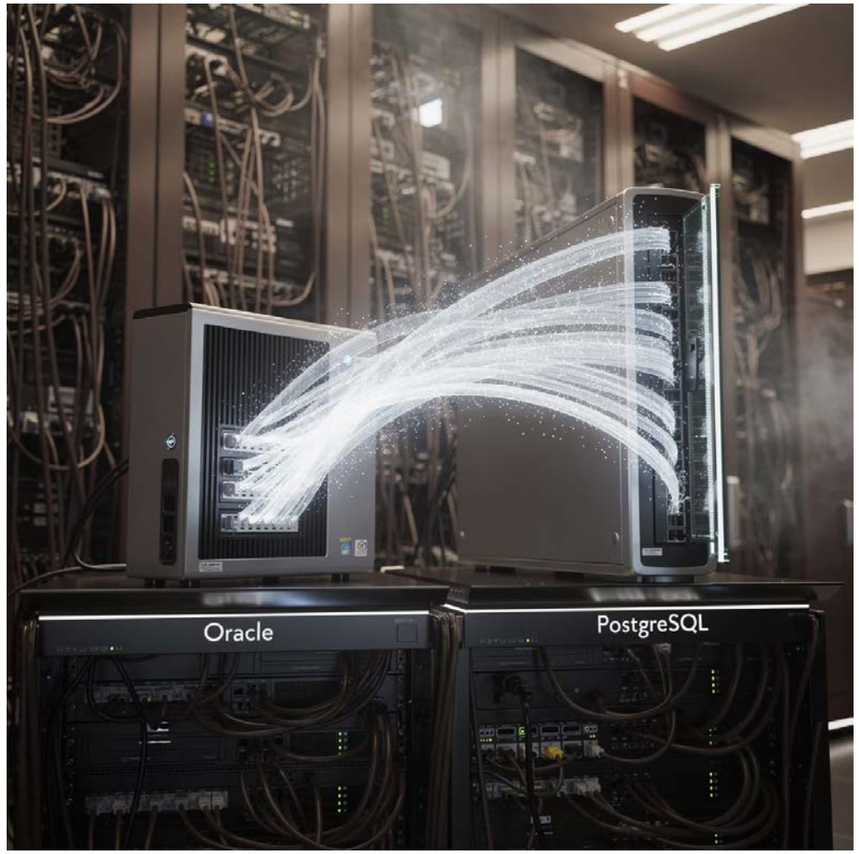
Database Modernization

Transitioned from Oracle to CloudSQL PostgreSQL using specialized migration tooling

Network Security

Implemented software-based network policies with protocol-aware filtering

The Database Migration Challenge



Oracle to PostgreSQL

The database migration proved to be one of the most complex aspects of Phase 2. Oracle's proprietary features required careful handling during the transition to CloudSQL PostgreSQL.

Key Migration Elements

- Oracle packages and stored procedures requiring rewriting
- Sequence generators and identity column mappings
- Date-time data type conversions and timezone handling
- Query optimization for PostgreSQL's execution planner

The team leveraged a specialized vendor-supported migration tool to automate much of the conversion while maintaining data integrity throughout the process.

Securing the Container Network

Software-Based Network Policies

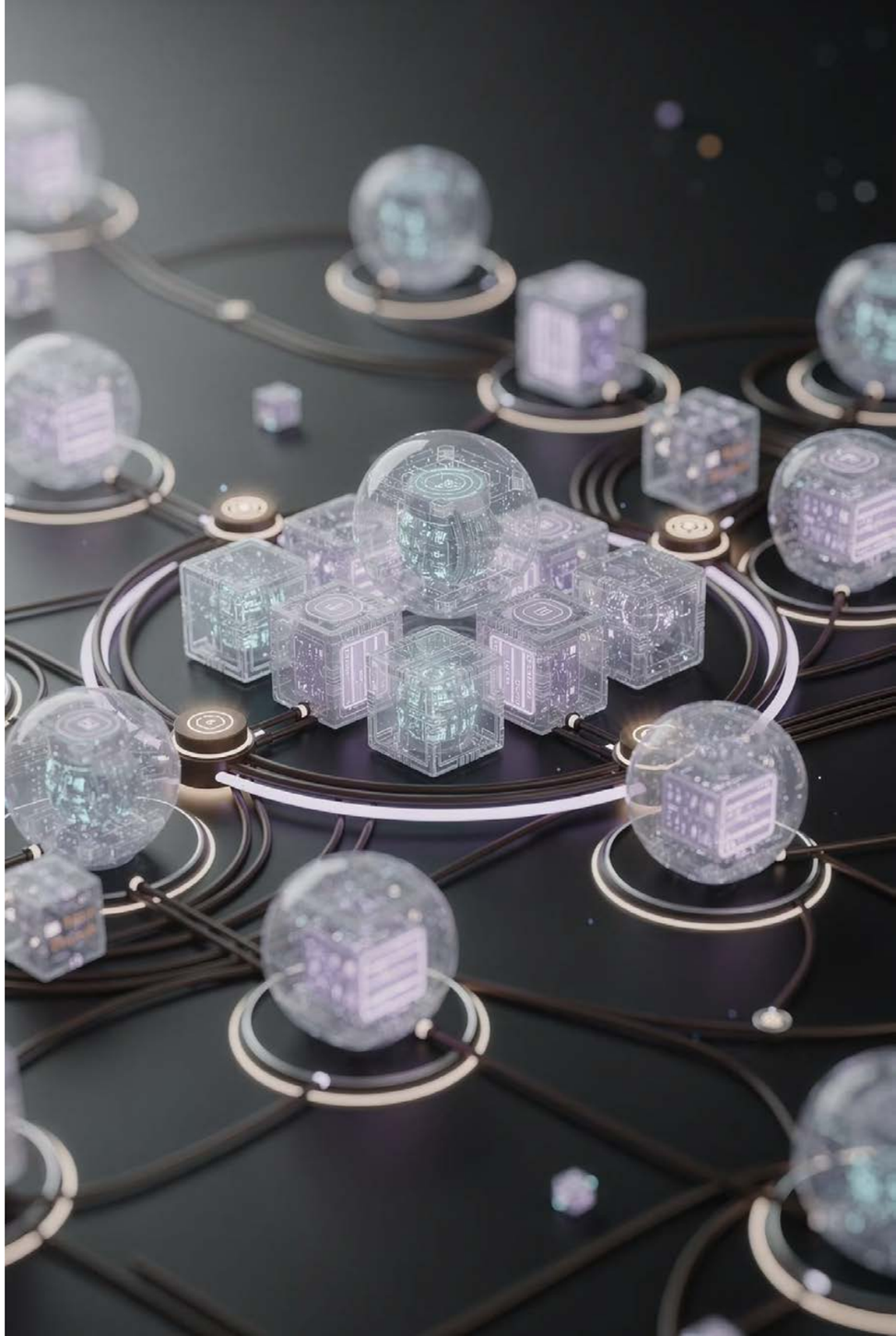
Implemented fine-grained network security controls using Kubernetes NetworkPolicy resources

Protocol-Aware Filtering

Applied layer 7 filtering rules to control application-level communication patterns

Pod Communication Control

Managed security across more than two hundred pod-to-pod communication paths with explicit allow/deny rules



Disaster Recovery Evolution

Legacy State

Basic backup procedures with limited testing and uncertain recovery times

1

Phase 1

Cloud-based backups with snapshot capabilities and documented recovery procedures

2

Phase 2

Multi-region deployment model with automated failover and regular full-scale recovery exercises

3

Key Lessons: Sequencing Migrations

What Worked

- Separating infrastructure migration from architectural changes reduced risk and complexity
- Establishing performance baselines in Phase 1 provided clear success criteria for Phase 2
- Parallel deployment approaches allowed thorough testing before cutover
- Incremental rollout of containerized services enabled rapid rollback if issues emerged

What We'd Do Differently

- Start database schema simplification earlier in the process
- Invest more upfront in observability tooling before migration begins
- Create more detailed runbooks for operational teams before go-live
- Schedule more buffer time for security validation in containerized environments

Operating Containerized Workloads

Resource Management

Setting appropriate CPU and memory limits required extensive profiling and tuning. Start conservative and optimize based on actual usage patterns.

Deployment Strategy

Implement rolling updates with health checks and automatic rollback. Blue-green deployments add safety but increase resource requirements.

Observability

Distributed tracing becomes essential for troubleshooting. Use Cloud logging and monitoring for logging aggregation and metrics collection across the container fleet.

State Management

Separate stateful from stateless workloads. Use StatefulSets for databases and message queues, Deployments for application tiers.

Implementing Cloud-Native Security

Identity and Access

Service accounts with minimal required permissions, regular credential rotation, and workload identity federation

Network Controls

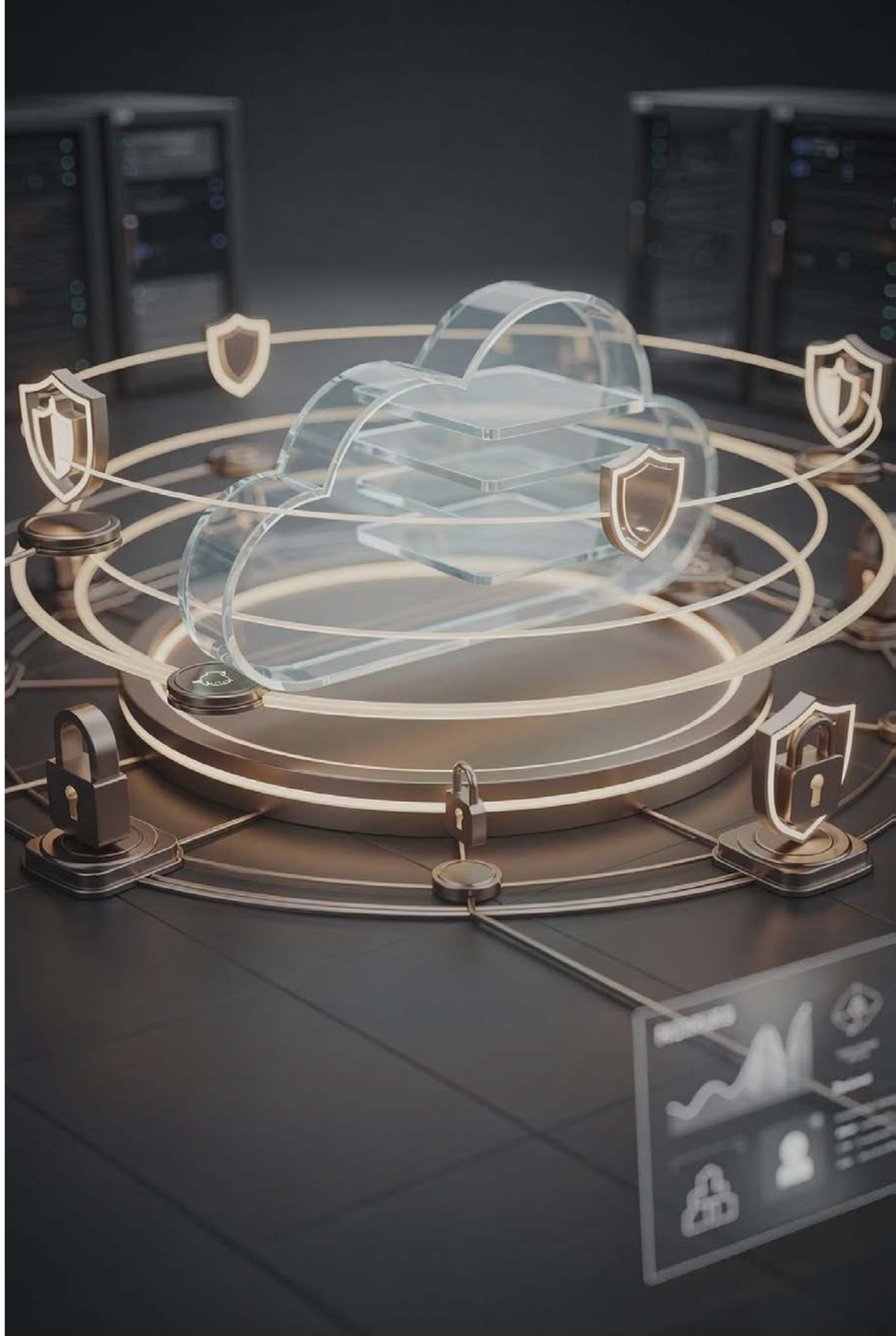
Network policies enforcing zero-trust pod communication, egress filtering, and private cluster endpoints

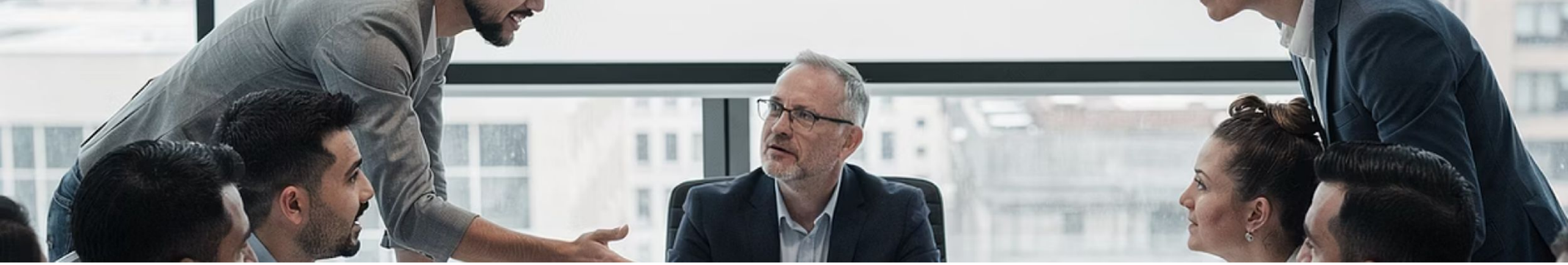
Container Security

Image scanning in CI/CD pipelines, signed images, pod security policies, and runtime threat detection

Data Protection

Encryption at rest and in transit, secrets management with rotation, and secure configuration injection





Practical Takeaways for Your Journey

Start with Dependency Mapping

Invest time understanding your current system architecture, integration points, and failure modes before planning the migration

Measure

Establish comprehensive baseline metrics before migration and monitor continuously during and after transition

Plan for Database Complexity

Database migrations typically take longer than expected. Budget extra time for testing, validation, and performance tuning

Test Disaster Recovery

Regular full-scale recovery exercises are the only way to validate your DR plan actually works under pressure

Thank you!

Questions?

Harris Peter Baskaran | Google Inc., USA | Conf42 DevOps 2026.