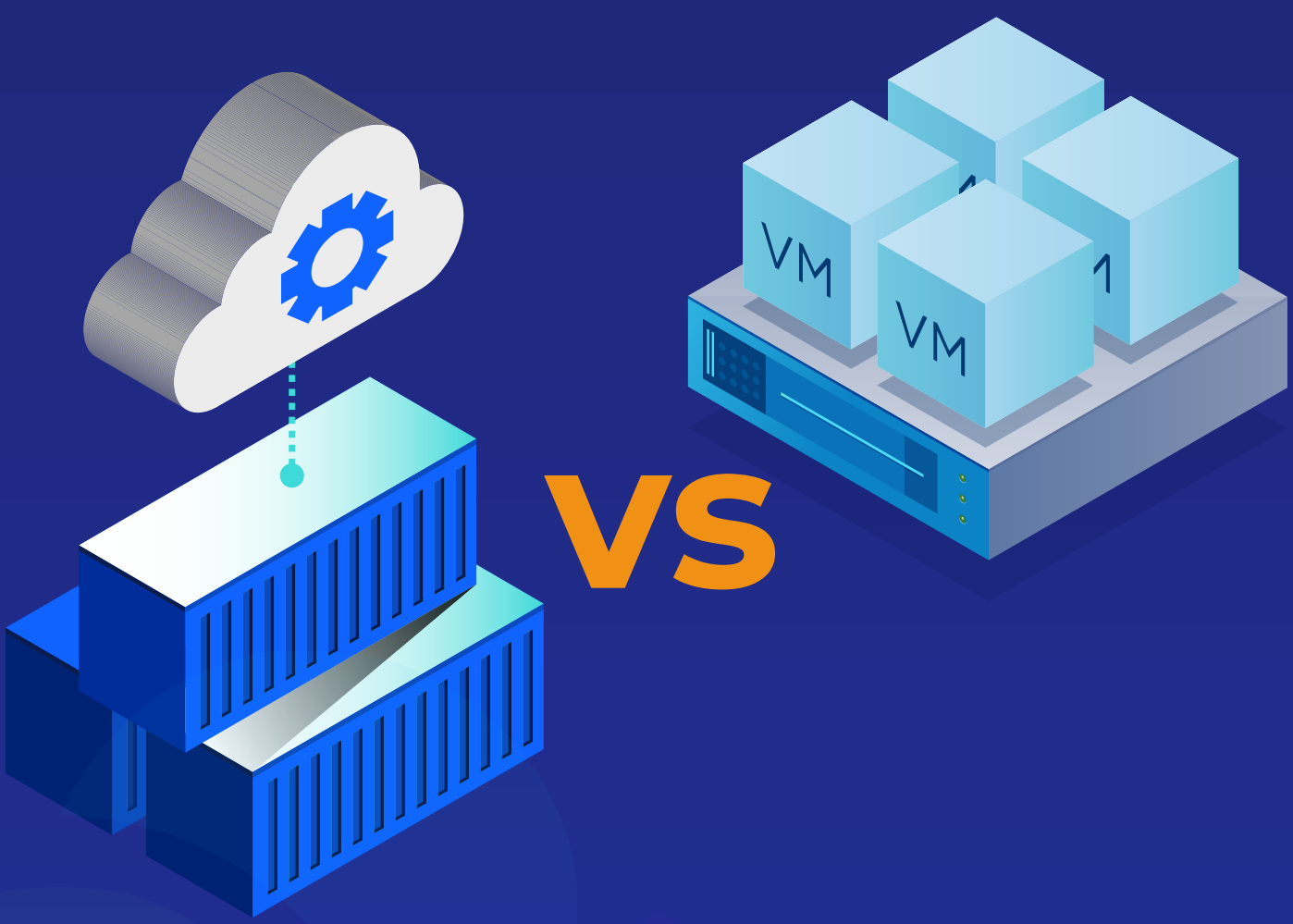
Naviteq

A Step-by-Step Guide to Migrating from Virtual Machines to Containers



Naviteq

A Step-by-Step Guide to Migrating from Virtual Machines to Containers

The application deployment landscape has undergone a significant shift, with containerization emerging as a transformative approach. Virtual Machines (VMs), while being effective, often come with certain challenges and limitations. Containers, on the other hand, offer a lightweight, portable and efficient approach to deploy applications at scale, enabling organizations to move at a much faster pace and iterate quickly.

Recently, organizations have been migrating to containers due to various reasons, including:

More efficient resource utilization

Rapid deployment and scaling

Improved application portability

Easy to setup across environments

Naviteq

Understanding VMs and containers

What are VMs?

VMs are software-defined environments that emulate physical computers. They operate by creating an isolated, virtualized layer on top of a physical host machine's hardware, allowing multiple virtual computers to run independently on a single physical host. This virtualization is managed by a hypervisor which allocates and manages the host's physical resources like CPU, memory, storage, and network interfaces among the virtual machines. Each virtual machine runs its own operating system and applications. It is completely unaware that it's sharing hardware resources with other VMs. This isolation ensures that problems or crashes in one VM don't affect others running on the same host.

What are containers?

Containers are lightweight, portable units designed to package an application along with all its dependencies, configurations, and libraries. This enables consistent deployment across different computing environments. Unlike traditional virtual machines, containers share the host operating system's kernel, making them more efficient and faster to start up. This approach to application packaging has revolutionized software deployment by ensuring that applications run reliably when moved from one computing environment to another, whether it's locally on a developer's laptop, a test environment or staging/production environment on the cloud. Containers achieve this consistency by bundling everything needed to run the application – code, runtime, system tools, system libraries, and settings – into a single, self-contained package.

Naviteq

VMs versus containers

Characteristic	VMs	Containers
Portability	Limited	Excellent
Scalability	Slow	Rapid
Isolation	High	Limited
Resource utilization	Less efficient	Very efficient
Boot time	Slower	Quicker

Naviteq

Pre-migration considerations

1. Assess current applications

Identify applications suitable for containerization, considering factors like statefulness, dependencies, and complexity.

Some key points are:

- Stateless services are prime candidates
- Micro-services architectures migrate easily
- Legacy monolithic applications may require significant refactoring
- Applications with complex OS-level dependencies need careful evaluation

2. Set objectives

Define clear goals for the migration. These can be improved scalability, faster deployment, or reduced operational costs.

Some key objectives to keep in mind are:

- Reduce infrastructure costs
- Improve deployment speed
- Enhance scalability
- Increase development team productivity
- Simplify infrastructure management

3. Choose the right tooling

Select tools like Docker for containerization, Kubernetes for orchestration, and Helm for package management.

- **Docker:** Container creation and management
- **Kubernetes:** Container orchestration
- **Helm:** Package management for Kubernetes
- **Docker Compose:** Local container development
- CI/CD tools (Jenkins, GitLab CI, GitHub Actions)

Naviteq

Preparing the environment

1. Infrastructure and dependencies

Ensure that your infrastructure can support containerized workloads, including sufficient CPU, memory, and network bandwidth.

Some important requirements are:

- Sufficient compute resources
- Container-optimized operating systems
- Network configuration supporting container networking
- Storage solutions compatible with container persistent storage

2. Select a container runtime

Choose a runtime like Docker or containerd to execute container images.

The most commonly used container runtimes are:

- **Docker:** Most popular, industry leader and feature-rich
- **Containerd:** Lightweight, CNCF-graduated runtime
- **CRI-O:** Kubernetes-focused runtime
- **Podman:** Daemonless alternative to Docker

3. Set up orchestration with Kubernetes

Deploy Kubernetes to manage and scale containerized applications. The process of setting up container orchestration using Kubernetes is as follows:

1. Choose managed or self-hosted Kubernetes
2. Configure cluster networking
3. Set up persistent storage
4. Implement role-based access control (RBAC)
5. Configure monitoring and logging

Naviteq

Step-by-step migration process

Step 1: Plan the migration

Identify applications to prioritize and create a detailed migration plan.

A. Create a comprehensive inventory of existing applications

Document all applications with details critical for containerization, such as their runtime environments, base operating systems, configuration files, environment variables, storage requirements, and network dependencies. Note which applications are already using technologies that work well with containers (stateless services or microservices) versus those that might need refactoring (monoliths with local file system dependencies). Also identify applications that use specific ports, require special hardware access, or have strict performance requirements.

B. Categorize applications by complexity and migration difficulty

Assess each application's containerization readiness by examining: whether they can run in isolated environments, their state management requirements, external dependencies, build processes, and deployment procedures. For example, stateless web applications are typically easier to containerize than applications that require shared file systems.

C. Prioritize applications with minimal dependencies

Start with applications that are naturally container-friendly. Generally, newer applications that are built with modern frameworks that are stateless and have well-defined API interfaces. These include - web services, static content websites, API based backend services etc.

D. Create a phased migration roadmap

Develop a timeline that includes container-specific milestones like:

- Setting up a container orchestration platform like Kubernetes.
- Establishing container registry and image management processes.
- Creating base container images and standardizing Dockerfile templates.
- Implementing container security scanning, monitoring and logging.
- Setting up container monitoring and logging.
- Training teams on container technologies and best practices. Each phase should include testing container configurations, validating application performance in containers, and ensuring proper resource allocation.

Naviteq's DevOps services, including Kubernetes management, can help you significantly in this regard and streamline your containerization efforts. By leveraging Naviteq's expertise, your team can gain the necessary skills to successfully adopt container technologies and achieve significant benefits in terms of agility, scalability, and efficiency.

Naviteq

Step 2: Create container images

Use Dockerfiles to define the environment and dependencies for each application.

A. Use multi-stage Dockerfiles for efficient builds

Multi-stage builds separate the build environment from the runtime environment, significantly reducing final image size.
Structure your Dockerfile with multiple FROM statements where:

- The first stage contains all build tools, SDKs, and dependencies needed for compilation.
- Subsequent stages copy only the necessary artifacts from the build stage.
- The final stage includes runtime dependencies and application binaries. For example, a Java application might use Maven in the build stage but only need a JRE in the runtime stage. This approach can reduce image size significantly and remove potential security vulnerabilities from build tools.

B. Minimize image size

Keep images lean by implementing several key practices:

- Remove unnecessary files, caches, and temporary data in the same layer where they are created.
- Use **.dockerignore** to exclude irrelevant files from the build context.
- Chain RUN commands with **&&** to reduce layer count.
- Clean package manager caches **(apt-get clean, rm -rf /var/cache/yum).**
- Use minimal slim or alpine base images when appropriate.
- Only install required packages and dependencies.
- Remove build dependencies in multi-stage builds.

C. Implement best practices for image security

Focus on building secure images from the ground up. Use the following best practices:

- Run containers as non-root users.
- Set filesystem and volume permissions appropriately.
- Regularly update base images to patch security vulnerabilities.
- Use COPY instead of ADD to prevent remote file injection.
- Specify exact versions of base images and dependencies.
- Implement resource limits and constraints.
- Scan images for known vulnerabilities using tools like Trivy or Snyk.
- Use proper secrets management solutions.

D. Use tools like Kaniko for secure image building

Kaniko offers several advantages for secure container builds. Its offerings are:

- Builds images in userspace without requiring Docker daemon.
- Runs in unprivileged containers.
- Supports building images directly to remote registries.
- Integrates well with CI/CD pipelines.
- Provides cache support for faster builds.
- Ensures consistent and reproducible builds.
- Offers detailed build logging and debugging capabilities.

E. Leverage official base images when possible

Official base images provide several benefits:

- Regular security updates and patches.
- Well-documented and maintained.
- Verified builds with known provenance.
- Optimized for common use cases.
- Consistent across environments.

Step 3: Test containerized applications

Thoroughly test containerized applications in a testing and staging environment to identify and address issues proactively.

A. Unit testing containerized components

Test individual components within containers to verify isolated functionality. Some things to keep in mind are:

- Write tests that can run within the container environment.
- Verify component behavior with mocked dependencies.
- Test configuration management and environment variable handling.
- Validate container startup scripts and initialization processes.
- Check error handling and logging mechanisms.
- Ensure proper resource cleanup on container shutdown.

B. Integration testing in staging environment

Focus on testing how containerized applications work together in a staging environment. Some key points to keep in mind are:

- Test service discovery and inter-container communication.
- Verify container orchestration behaviors (scaling, failover, rolling updates).
- Test network policies and service mesh configurations.
- Validate persistent storage interactions.
- Check load balancing and service routing.
- Test container restart and recovery scenarios.
- Ensure proper secrets management and configuration injection.

C. Performance benchmarking

Measure and compare performance metrics between containerized and non-containerized versions of the application. Some performance indicators to check for:

- Response time and latency under different load conditions.
- Resource utilization - CPU, memory, network and disk I/O.
- Container startup and scaling times.
- Database connection pooling effectiveness.
- Network throughput between containers.

D. Security scanning of container images

Implement comprehensive security checks:

- Scan base images and dependencies for known vulnerabilities.
- Check for sensitive data exposure in image layers.
- Verify compliance with security policies.
- Test container runtime security settings.
- Validate network security policies.
- Verify secure configuration of container runtime Use tools like Trivy, Clair for automated scanning.

E. Compatibility testing across different environments

Ensure consistent behavior across different platforms and configurations. To ensure this follow the points mentioned below:

- Test on different container orchestration platforms like Kubernetes, ECS etc.
- Verify functionality across different cloud providers if applicable.
- Test with different storage classes and volume types.
- Validate networking in different subnet configurations.
- Check compatibility with different monitoring solutions.
- Ensure consistent logging across platforms.

Step 4: Implement orchestration with Helm

Use Helm charts to package and deploy applications to Kubernetes. To implement orchestration with Helm, you'll first need to create a chart structure which includes a Chart.yaml file that defines metadata, a values.yaml file for configurable parameters, and templates directory containing Kubernetes manifest templates.

The process of creating a Basic Chart Structure is as follows:

1. Initialize a new chart with **helm create chartname.**
2. Organize files into required directories **(templates/, charts/).**
3. Define chart metadata in Chart.yaml including dependencies.
4. Create default values in **values.yaml.**
5. Set up helpers in **_helpers.tpl.**
6. Include **README.md** with usage instructions and configuration details.

Step 5: Deploy to production

Carefully deploy containerized applications to production, monitoring their performance and health. You can use the following deployment strategies:

A. Use blue-green or canary deployment strategies

Implement progressive deployment patterns to minimize risk and downtime. In blue-green deployments - maintain two identical environments (blue and green) where one serves production traffic while the other receives updates. Switch traffic gradually between environments using service mesh or ingress controllers. For canary deployments, release new versions to a small subset of users first e.g., 5-10% of traffic. Monitor this change for issues and gradually increase traffic to the new version. Define clear metrics and thresholds for automated rollback triggers.

B. Implement robust monitoring

Set up comprehensive monitoring across all layers of the containerized infrastructure. Deploy Prometheus for metrics collection and Grafana for visualization. Monitor key metrics like container resource usage, application response times, error rates, and business KPIs. Create custom dashboards for different stakeholder needs DevOps engineers, IT Managers etc. Implement alerting with proper thresholds and escalation paths and set up distributed tracing with tools like Jaeger to track requests across services.

C. Configure auto-scaling

Establish horizontal pod autoscaling (HPA) based on custom metrics and resource utilization. Define scaling policies that account for both application-specific metrics like requests per second, queue length etc. and resource metrics like CPU, memory usage etc. Set appropriate minimum and maximum replica counts based on application requirements and implement cluster autoscaling to handle pod scheduling demands. Lastly, test scaling behavior under various load conditions to verify proper operation.

D. Set up centralized logging

Implement a unified logging solution using tools like Elasticsearch, Fluentd etc. To ensure effective log management in your containerized environment, implement a comprehensive log aggregation strategy. Centralize logs from all containers and cluster components to a unified logging platform. Establish robust log retention policies and configure log rotation to optimize storage usage. Standardize log formats across applications to facilitate parsing and analysis. Implement log-based alerting for critical events to enable timely response. Prioritize log security by enforcing strict access controls. Lastly, consider compliance requirements for log retention and the handling of sensitive data.

E. Establish rollback mechanisms

Create comprehensive rollback procedures for all deployment components. Maintain versioned copies of all configurations and container images. Implement automated rollback triggers based on monitoring metrics and document rollback procedures clearly, including manual intervention steps if needed. Lastly, configure your continuous delivery pipeline to support rapid rollbacks positively.

Key challenges and solutions

Some key challenges and solutions with the migration and containerizing applications are:

Handling state and storage

Handling state and storage is tricky while migrating to containerized applications. These challenges can be resolved by using the following approaches:

Persistent volumes

Use persistent volumes for storing application data and design applications to be stateless whenever possible. Persistent volumes provide a critical solution for maintaining application data integrity during container migrations. By decoupling storage from the container lifecycle, organizations can ensure data persistence even when individual containers are destroyed and recreated. This approach allows stateful data to be preserved across container restarts and node migrations, enabling more flexible and reliable application deployment.

Using stateful sets for stateful applications

Kubernetes StatefulSets offer specialized handling for stateful applications that require stable, unique network identifiers and persistent storage. Unlike standard deployments, StatefulSets maintain a stable hostname and persistent storage volume for each pod, ensuring consistent state management for databases, message queues, and other stateful services.

Using external storage providers

External storage providers enable organizations to leverage specialized storage solutions that integrate seamlessly with containerized environments. Cloud-native storage platforms like Amazon EBS, Google Persistent Disk, and Azure Disk Storage offer scalable, high-performance storage options that can be dynamically provisioned and attached to containers. These providers support features like automatic volume creation, dynamic provisioning, snapshot management, and cross-zone replication.

Using database-specific migration approaches

Migrating databases to containerized environments requires tailored strategies that address each database system's unique characteristics. For relational databases like PostgreSQL and MySQL, approaches include using official container images, implementing custom initialization scripts, and configuring persistent volumes for data storage. NoSQL databases like MongoDB and Cassandra benefit from StatefulSet configurations that maintain data consistency and node identity.

Security considerations

Implement robust security measures, including image scanning, secret management, and network security. These challenges can be resolved by using the following approaches:

Scan container images for vulnerabilities

Container image scanning is a critical security practice that systematically identifies potential security risks and vulnerabilities within container images before deployment. Automated scanning tools like Clair, Trivy, and Anchore analyze images against comprehensive vulnerability databases, checking for known security issues, outdated dependencies, and potential exploits. These tools provide detailed reports highlighting critical vulnerabilities and recommend remediation steps.

Use minimal base images

Utilizing minimal base images significantly reduces the attack surface and potential vulnerabilities in container deployments. Lightweight images like Alpine Linux, Distroless, and Ubuntu Minimal provide essential functionality with minimal additional packages and system components.

Implement secrets management

Effective secrets management ensures sensitive information like credentials, API keys, and configuration data are securely stored and accessed. Kubernetes offers native secrets management through Secret objects, while external tools like HashiCorp Vault provide advanced encryption and access control.

Configure network policies

Network policies provide granular control over container communication, defining allowed ingress and egress traffic patterns. Kubernetes Network Policies enable administrators to isolate container namespaces, control pod-to-pod communication and restrict external network access.

Use runtime security tools

Runtime security tools provide real-time monitoring, threat detection, and protective measures for containerized environments. Solutions like Falco offer continuous container monitoring, anomaly detection and integrations with SIEM systems.

Naviteq

Managing complexity with Helm and GitOps

Utilize Helm and GitOps tools like ArgoCD to automate deployment and configuration management.

Helm for package management

Helm serves as a powerful Kubernetes package manager, enabling organizations to define, install, and upgrade complex Kubernetes applications. It provides a powerful mechanism for creating standardized, reusable Kubernetes application deployments by offering comprehensive features like - managing resource dependencies, supporting extensive parameterization and configuration customization, and facilitating advanced version management with robust rollback capabilities. Through sophisticated templating, Helm simplifies the deployment of complex applications, while simultaneously providing a centralized repository infrastructure that enables efficient sharing and distribution of application configurations across development teams and organizations.

ArgoCD for GitOps workflows

ArgoCD implements declarative, Git-based continuous delivery for Kubernetes that enables automated synchronization between Git repositories and cluster states. ArgoCD revolutionizes Kubernetes deployments through automated manifest synchronization and version-controlled infrastructure management which provides comprehensive support for multiple cluster configurations with robust rollback and history tracking capabilities. The platform offers intuitive visualization of application deployment status and integrates seamlessly with existing CI/CD pipelines.

Prometheus for monitoring

Prometheus provides robust, open-source monitoring and alerting for containerized environments. Prometheus provides robust monitoring for containerized environments by collecting and storing time-series metrics through multi-dimensional data collection. It has a powerful query language (PromQL), and generates real-time alerts and notifications. Its capabilities include providing comprehensive and flexible monitoring solutions for complex distributed systems.

Grafana for visualization

Grafana offers advanced observability and visualization capabilities for metrics collected from various sources. It provides interactive, customizable dashboards that support multiple data source integrations, enabling complex data visualization and analysis. Its features include implementing advanced querying and transformation techniques, generating real-time alerts and notifications, and providing robust role-based access control.

Post-migration best practices

The following are some recommended post-migration best practices:

Versioning and tagging containers

Use semantic versioning to manage container image versions.
Some best practices are:

- Implement immutable infrastructure principles where once a service is deployed, it's never modified. This makes rollbacks simpler, increases reliability and improves security of the deployments.
- Create a well-structured tagging strategy to efficiently organize and retrieve resources. This includes - consistent naming convention, descriptive tags for resource categorization, designing tagging conventions that can be modified as the organization grows, regularly review and update tagging strategies etc.
- Automate version management to streamline the release process by automating tasks like - version number determination, changelog generation and package publishing.

Monitoring and ongoing optimization

It's a good practice to continuously monitor and optimize containerized applications to ensure that they are running efficiently.

- Regularly perform audits to detect any anomalies in the systems.
- Focus on resource utilization analysis to optimize deployment and to keep resource wastage in check.
- Keep rightsizing applications to ensure every container has the right amount of CPU and memory to perform efficiently without being under or over provisioned.
- Always try to stay up-to-date with container ecosystem developments.

Naviteq

Conclusion

The migration from virtual machines to containers is a strategic move that can significantly enhance an organization's IT infrastructure. By adopting containerization technologies, businesses can streamline application deployment, improve scalability, and boost operational efficiency.

However, such a transition requires careful planning and execution. It involves a comprehensive assessment of applications, selection of suitable containerization platforms, and implementation of robust orchestration strategies. Leveraging a managed platform like Naviteq can greatly simplify this process. With a team of experts and a suite of services including DevOps as a Service, Kubernetes Management, and CI/CD pipelines, Naviteq can help organizations accelerate their containerization journey. By offloading the complexities of container orchestration and management, businesses can focus on their core competencies and drive innovation.

This migration process involves several critical considerations. While it is recommended that you leverage support from industry professionals such as Naviteq, here's a short checklist you can follow:

- **Assess applications:** Identify suitable applications, evaluate dependencies, and assess performance impact.
- **Select a platform:** Choose a containerization platform like Docker, considering factors like scalability, security, and ease of management.
- **Refactor applications:** Address dependencies, break down monolithic applications into microservices, optimize configurations, and package applications into container images.
- **Implement orchestration:** Select an orchestration tool like Kubernetes, define deployment and scaling strategies, configure network and storage solutions, and implement monitoring and logging.

- **Prioritize security:** Implement robust security measures, secure container images and registries, enforce access controls, and regularly update and patch.
- **Test and deploy:** Thoroughly test containerized applications, establish a CI/CD pipeline, and develop strategies for production rollout.
- **Monitor and manage:** Implement monitoring tools, establish incident response procedures, and continuously optimize container configurations and resource utilization.

Naviteq

Transform Your DevOps with Naviteq

Ready to optimize your DevOps processes and drive efficiency? Naviteq's experts specialize in streamlining application deployment, improving scalability, and implementing cutting-edge tools like Kubernetes, Helm, and CI/CD pipelines.

Whether you're refining your workflows or starting fresh, we'll help you build agile, secure, and scalable systems that support your growth. Let Naviteq simplify the complexities of DevOps so you can focus on innovation.

Contact us today to explore tailored solutions that align with your goals and set you up for long-term success!