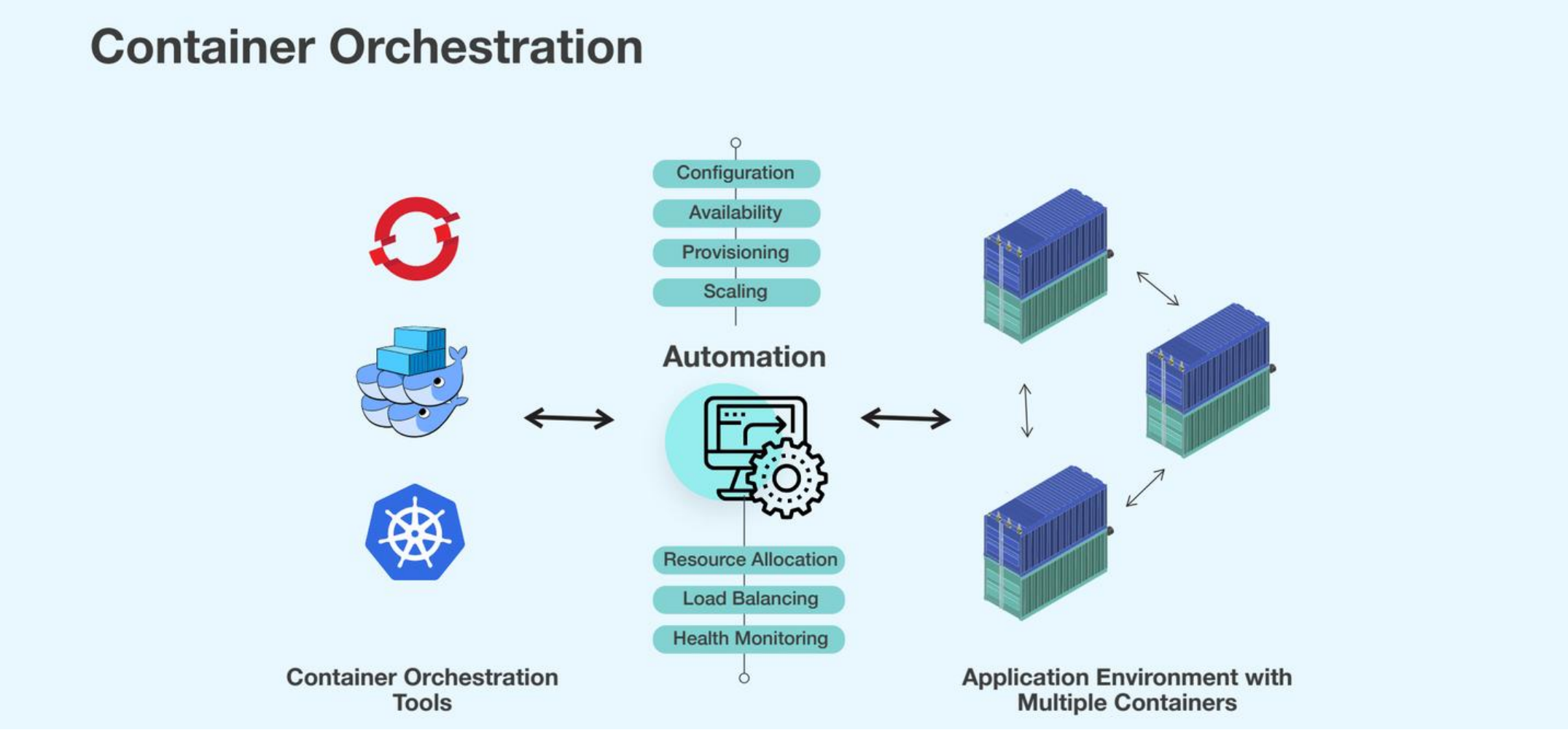


Containers: Powering Modern Application Development

Introduction to Containers

- **Lightweight, Portable Packaging**
  Containers encapsulate applications and dependencies into a single executable unit, ensuring consistent runtime across environments.
- **Isolated and Efficient**
  Containers provide isolation from other applications, while sharing the host kernel for better resource efficiency compared to VMs.
- **Reproducible and Portable**
  Container images guarantee the same application version and dependencies are deployed consistently across environments.
- **Podman: Daemonless Container Engine**
  Podman is a container engine that allows developers to run and manage containers without a central daemon, improving security and integration with CI/CD pipelines.
- **Building Custom Images**
  Containerfiles (Dockerfiles) define the steps to build custom container images, ensuring reproducibility and enabling deployment pipelines.
- **Persisting Data in Containers**
  Configuring persistent storage using bind mounts or named volumes ensures data survives container restarts or recreations.

How containers facilitate application development

- **Consistent Dev-to-Prod Workflow**
  Developers build and test against the same container image that will run in production, reducing "it works on my machine" issues.
- **Rapid Iteration**
  Containers can be started and stopped in seconds, allowing developers to spin up isolated environments for feature development or troubleshooting without waiting for VM provisioning.
- **Modularization**
  By breaking a monolithic application into multiple containers (each handling a specific function), teams can develop, test, and deploy components independently.
- **Dependency Management**
  All required libraries, runtime versions, and configurations are defined in the image, eliminating version drift and simplifying onboarding of new team members.
- **Scalability**
  Containers can be replicated horizontally to handle increased load, and orchestration platforms (e.g., Kubernetes, OpenShift) manage scaling automatically.

Podman: Daemonless Container Engine

- ## Introduction to Podman

  Podman is a daemonless container engine that is compatible with the Open Container Initiative (OCI) specifications. Unlike Docker, Podman does not require a long-running background service, and each command interacts directly with the container runtime. Podman offers a rootless mode, allowing developers to run containers without root privileges, reducing the attack surface.

- ## Core Operations in Podman

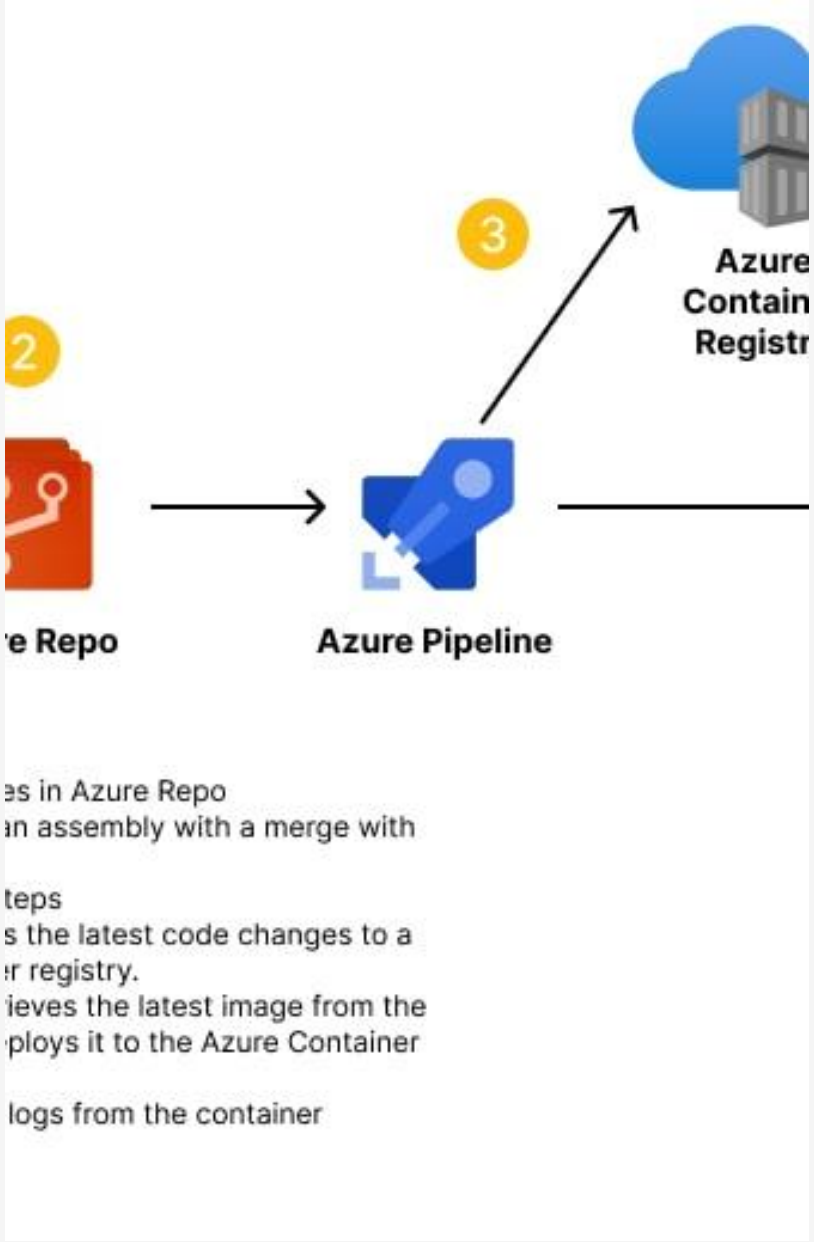
  Podman provides a set of core operations for managing containers and images, including viewing available containers and images, running a container with various options, and managing the container lifecycle.

- ## Pods in Podman

  Podman introduces the concept of pods, which are groups of one or more containers that share networking and can share storage. Pods are created using `podman pod create` and containers can be run within them.

- ## Advantages of Podman

  Podman offers several advantages over other container engines, including rootless execution for enhanced security, Docker CLI compatibility, and Kubernetes YAML support for generating pod definitions from running containers.



Container Images and Registries

Container images serve as the blueprint for containers, comprising a filesystem snapshot, application binaries, configuration files, runtime libraries, and metadata. These images are composed of layers, where each instruction in a containerfile creates a new layer, allowing for efficient reuse and caching. Container registries, both public and private, provide a way to store and manage these container images.

Container Images and Custom Builds

- **Container Images**
  Read-only templates with filesystem snapshot, binaries, configs, and metadata
- **Image Structure and Layers**
  Layers are stacked in a union filesystem for efficient reuse and caching
- **Container Registries**
  Public (Docker Hub, Quay.io) and private (self-hosted, cloud-hosted) registries
- **Searching and Pulling Images**
  Search registries, pull images by tag, inspect metadata
- **Custom Container Images**
  Encapsulate application runtime, dependencies, and configuration

Building Custom Container Images

Containerfile Fundamentals

The Containerfile (formerly Dockerfile) is the blueprint for building custom container images, containing instructions like FROM, RUN, COPY, ENV, WORKDIR, ENTRYPOINT, and CMD.

Build Context and Caching

The build context is the directory and subdirectories sent to the container runtime during the build process. Layer caching is a crucial optimization, where unchanged layers from previous builds are reused to accelerate the build process.

Best Practices for Custom Images

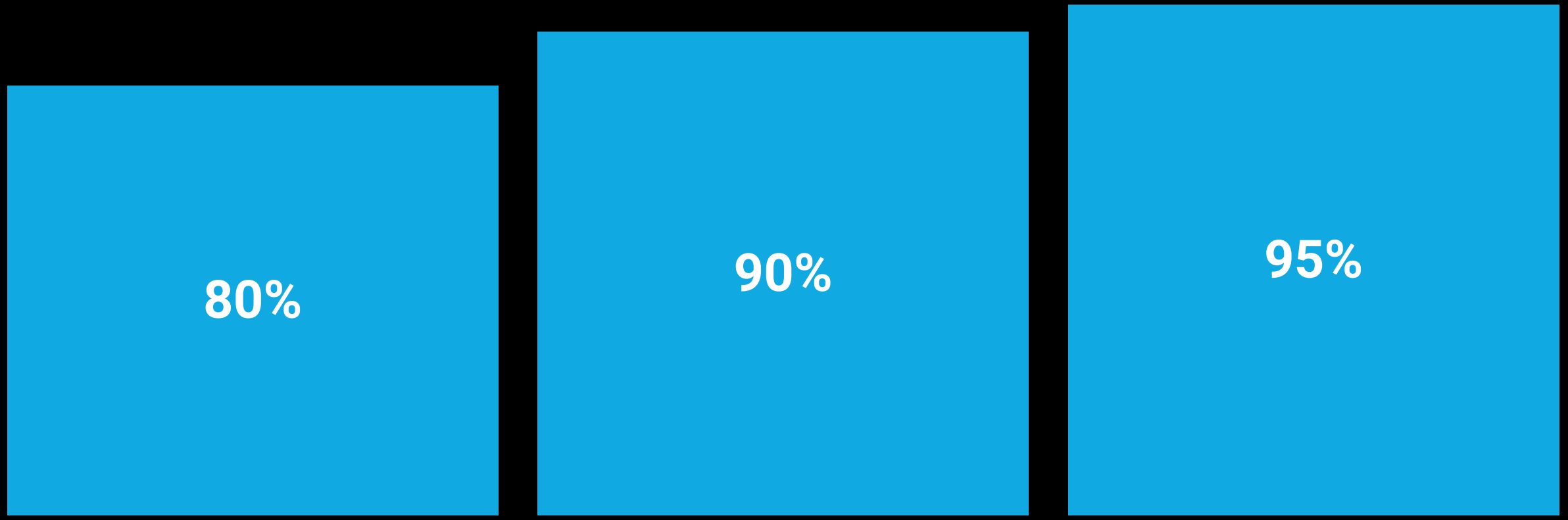
Minimize layers, order instructions to leverage effective caching, use slim base images, avoid storing secrets in the image, and include health checks and metadata for traceability and automated scanning.

Continuous Integration Considerations

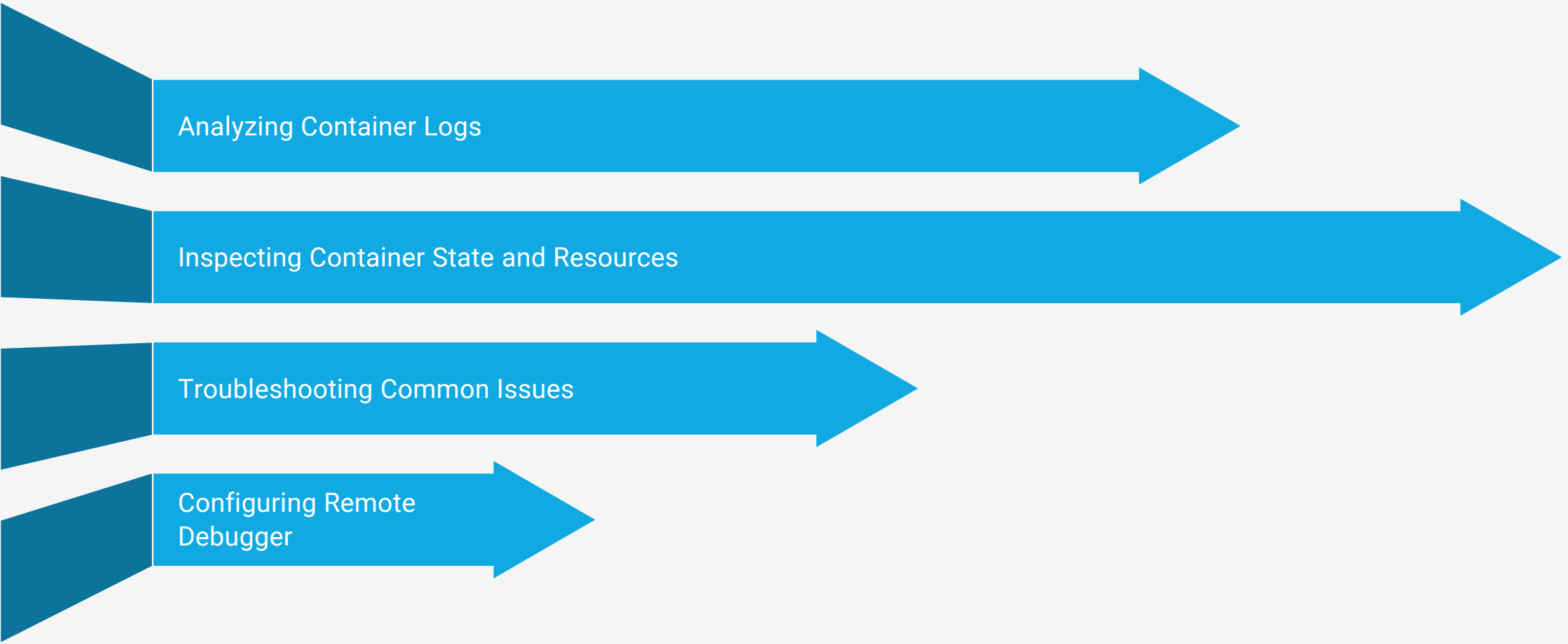
Automate image builds with CI pipelines, configuring the pipeline to automatically tag images based on Git commits or semantic version files. Integrate security scanners to identify vulnerabilities before pushing to a registry.

Persistent Storage for Stateful Services

Comparison of common persistent storage solutions for containers



Troubleshooting Containers



Orchestrating Containers with Kubernetes and OpenShift

Persistent Storage for Stateful Services

Explore options for managing persistent data in containers, such as bind mounts and named volumes. Discuss the importance of ensuring data persists beyond the container's lifecycle.

Scaling Databases Across Hosts

Examine strategies for scaling database containers across multiple hosts, including the use of network-attached storage solutions and cloud-provided storage services. Discuss the concept of shared persistent volume claims in Kubernetes and OpenShift.

Security Considerations

Highlight the importance of proper file permissions, ownership, and data-at-rest encryption when using persistent storage. Discuss the role of filesystem-level encryption and cloud-provided encrypted volumes.

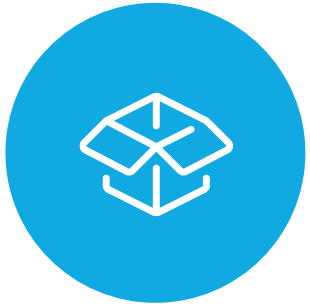
Kubernetes and OpenShift Orchestration

Introduce Kubernetes and OpenShift as container orchestration platforms, outlining key concepts such as clusters, pods, deployments, stateful sets, services, and ingress/routes. Discuss the similarities and differences between the two platforms, including authentication, security, and developer experience.

Container Orchestration with Kubernetes and OpenShift

- **Cluster**
  Set of worker and control-plane nodes
- **Pod**
  Smallest deployable unit - containers in a shared namespace
- **Deployment**
  Manages stateless workloads with rolling updates and rollbacks
- **StatefulSet**
  Manages stateful workloads like databases with stable network identities
- **Service**
  Abstraction exposing pods via stable IP and DNS
- **Ingress/Route**
  Provides external HTTP/HTTPS access with routing and TLS
- **ConfigMap and Secret**
  Separate configuration and sensitive data from container images

Containerized Application Lifecycle: From Fundamentals to AI-Powered Orchestration



Container Fundamentals

Understand the core concepts of containerization, including containers, images, and registries

Application Development

Learn the full development lifecycle for building containerized applications

Orchestration at Scale

Explore container orchestration platforms like Kubernetes to manage and scale containerized applications

AI-Infused Containerization

Discover how to leverage AI and machine learning to enhance every stage of the containerization process