Application Containerization Study Material for End Term

**UNIT IV: Development Environment**

**1. Overcoming Issues with Different Environments**

1. **Environment Drift**
   * **Definition**: Environment drift occurs when the development, testing, staging, and production environments become misaligned over time. This misalignment can lead to bugs or inconsistencies that appear only after deployment.
   * **Causes**:
     + Different OS or OS versions across machines (e.g., Ubuntu 18.04 vs. Ubuntu 20.04).
     + Library or dependency mismatches (e.g., Node.js v12 vs. Node.js v14).
     + Manual configuration changes that aren’t tracked in version control.
   * **Impacts**:
     + Increased debugging time, as an issue might only manifest in production.
     + Decreased confidence in the deployment process.
   * **Solutions**:
     + **Infrastructure as Code (IaC)**: Use tools like Terraform, AWS CloudFormation, or Pulumi to define and manage environment configurations in code.
     + **Configuration Management**: Tools like Ansible, Chef, or Puppet ensure consistent setup across all machines.
     + **Containerization**: Docker or other container technologies standardize the runtime environment.
2. **Version Control of Environment Configurations**
   * **Importance**:
     + Track changes over time, making it easier to roll back to a known good state.
     + Facilitate collaboration among team members.
   * **Implementation**:
     + Store scripts (shell scripts, PowerShell, Python) or Dockerfiles in a Git repository.
     + Use CI/CD pipelines to automatically apply or test configuration changes.
3. **Automation & Continuous Integration**
   * **Purpose**: Ensures that every change (code or configuration) is automatically tested in a controlled environment.
   * **Best Practices**:
     + Run automated tests (unit, integration, end-to-end) in identical containers or virtual machines.
     + Keep build pipelines versioned so that the same pipeline can be reproduced if needed.

**2. Development Environment**

1. **Local Development**
   * **Components**:
     + IDE or text editor (VSCode, IntelliJ, PyCharm, etc.).
     + Language runtimes and package managers (e.g., Node.js + npm/yarn, Python + pip).
     + Local databases (SQLite, MySQL, PostgreSQL) for quick testing.
   * **Best Practices**:
     + Use container-based development if possible (e.g., Docker Compose) to mirror production dependencies.
     + Maintain a consistent coding style with linters (ESLint, Pylint) and formatters (Prettier, Black).
2. **Remote Development**
   * **Scenarios**:
     + Large applications that require more resources than a local machine can provide.
     + Teams that want a centralized development environment for consistency.
   * **Tools**:
     + VSCode Remote Development or GitHub Codespaces.
     + SSH-based development on remote servers.
3. **Common Challenges**
   * **Dependency Hell**: Conflicts between different versions of libraries or frameworks.
   * **Performance**: Remote servers may introduce network latency; local machines may lack sufficient resources for large projects.
   * **Security**: Handling credentials and secrets securely, especially in shared environments.

**3. Testing Environment**

1. **Definition & Purpose**
   * A dedicated setup to run tests—functional, integration, regression, performance—before code moves to staging or production.
   * Mimics production as closely as possible but often uses test data or mock services.
2. **Types of Testing Environments**
   * **Integration Testing Environment**: Combines multiple services to ensure they work together correctly.
   * **Performance Testing Environment**: Focuses on load, stress, or endurance tests.
   * **Security Testing Environment**: Includes dynamic (DAST) or static (SAST) tools to identify vulnerabilities.
3. **Automation & Tooling**
   * **Continuous Testing**: Integrate test suites into CI pipelines.
   * **Popular Frameworks**:
     + **Web/UI**: Selenium, Cypress, Playwright.
     + **Backend**: JUnit (Java), PyTest (Python), Jest/Mocha (JavaScript).
     + **Security**: OWASP ZAP for DAST, SonarQube for static analysis.
4. **Environment Management**
   * **Infrastructure as Code** to provision testing environments on demand (e.g., ephemeral test environments in the cloud).
   * **Data Management**: Use anonymized or synthetic data to protect privacy while simulating realistic conditions.

**4. Staging Environment**

1. **Definition & Role**
   * A near-identical copy of production used for final verification.
   * Often includes realistic data sets (sanitized) to replicate production-like scenarios.
2. **Testing & Validation**
   * **User Acceptance Testing (UAT)**: Real users or business stakeholders verify features.
   * **Performance & Load Testing**: Confirms the system meets SLAs under expected traffic volumes.
   * **Deployment Rehearsals**: Practice rolling out new releases or updates to reduce production risks.
3. **Challenges**
   * **Cost**: Maintaining a staging environment that mirrors production can be expensive.
   * **Data Privacy**: Must ensure compliance with regulations (GDPR, HIPAA) if using real data.
4. **Best Practices**
   * Keep staging and production configurations as similar as possible (same OS, same container versions).
   * Automate deployments to staging using the same pipeline as production.

**5. Production Environment**

1. **Definition**
   * The live environment where end-users interact with the application or service.
   * Must be robust, secure, and have high uptime.
2. **Deployment Strategies**
   * **Blue-Green Deployments**:
     + Keep two identical environments (Blue and Green).
     + Users are routed to Blue while Green is updated. Then traffic is switched to Green, minimizing downtime.
   * **Rolling Updates**:
     + Update a subset of servers or containers at a time.
     + Gradually replace old versions with new ones, ensuring partial availability.
   * **Canary Releases**:
     + Release new features to a small portion of users or servers.
     + If no issues arise, gradually expand to more users.
3. **Monitoring & Observability**
   * **Metrics**: CPU, memory, disk I/O, request latency, error rates (collected via tools like Prometheus, Datadog).
   * **Logs**: Centralize and analyze using ELK (Elasticsearch, Logstash, Kibana) or Splunk.
   * **Tracing**: Distributed tracing (Jaeger, Zipkin) to pinpoint performance bottlenecks across microservices.
4. **Incident Response**
   * **On-Call Rotation**: Engineers available to handle alerts 24/7.
   * **Incident Management**: Clear procedures for triage, escalation, and communication.
   * **Post-Incident Reviews**: Root cause analysis and continuous improvement.

**6. Virtual Machines for Dev/Deployments**

1. **Virtualization Overview**
   * **Hypervisors**:
     + **Type 1 (Bare Metal)**: VMware ESXi, Microsoft Hyper-V, etc.
     + **Type 2 (Hosted)**: VirtualBox, VMware Workstation, etc.
   * **Resource Isolation**: Each VM runs its own OS, ensuring strong isolation between environments.
2. **Advantages**
   * **Isolation**: Misconfiguration or crashes in one VM do not affect others.
   * **Snapshots**: Quickly revert to a previous state if an update causes issues.
   * **Mature Ecosystem**: Tools and practices for VM management are well-established.
3. **Disadvantages**
   * **Overhead**: Each VM requires its own OS kernel, which can be resource-intensive.
   * **Slower Startup**: Boot times for full OSes are longer than containers.
   * **Scaling Complexity**: Managing multiple VMs can become cumbersome without orchestration tools.
4. **Use Cases**
   * Legacy applications that require full OS isolation.
   * Organizations that need separate OS-level environments for compliance or licensing.

**7. Docker for Dev/Deployments**

1. **Containers vs. VMs**
   * **Architecture**:
     + Containers share the host OS kernel, making them lighter and faster.
     + VMs each include a full guest OS.
   * **Performance**:
     + Containers have near-native performance with minimal overhead.
     + VMs can be heavier on CPU, RAM, and storage.
2. **Docker in Development**
   * **Docker Compose**:
     + Defines multi-container applications (e.g., web app + database + cache).
     + Provides a single command to spin up the entire environment.
   * **Benefits**:
     + Ensures every developer has the same environment.
     + Reduces “it works on my machine” issues.
3. **Docker in Production**
   * **Deployment Workflows**:
     + Build Docker images on a CI server, push them to a registry, then pull them onto production servers.
   * **Security Considerations**:
     + Use minimal base images (e.g., Alpine) to reduce attack surface.
     + Regularly update images to patch vulnerabilities.
4. **Best Practices**
   * **Use Official Base Images**: Typically well-maintained and vetted.
   * **Keep Images Small**: Remove unnecessary packages, cache busting layers properly.
   * **Immutable Infrastructure**: Treat containers as disposable; never manually patch a running container.

**8. Advantages and Drawbacks of Containerization**

1. **Advantages**
   * **Resource Efficiency**: Containers share the host kernel, using less memory and CPU overhead than VMs.
   * **Portability**: Build once, run anywhere Docker is installed.
   * **Scalability**: Quickly scale services up or down by launching or stopping containers.
   * **Faster Delivery**: Consistent, reproducible environments enable rapid deployment cycles.
2. **Drawbacks**
   * **Security Isolation**: Containers rely on kernel-level isolation, so a kernel exploit can affect all containers.
   * **Orchestration Complexity**: Managing many containers across multiple hosts requires orchestration tools (Kubernetes, Docker Swarm).
   * **Stateful Applications**: Containers are ephemeral, so special handling (volumes, external storage) is needed for persistent data.

**UNIT V: Docker Fundamentals and Internals**

**1. Docker Container States**

1. **Lifecycle Stages**
   * **Created**: Container is defined but not running.
   * **Running**: Container’s main process is active.
   * **Paused**: Process is suspended, preserving state in memory.
   * **Stopped/Exited**: Process has ended; container can be restarted if needed.
   * **Restarting**: Container is automatically restarting (e.g., due to a failure and restart policy).
2. **Commands & Usage**
   * docker ps or docker container ls: View running containers.
   * docker stop <container>: Gracefully stop a running container.
   * docker rm <container>: Remove a stopped container from Docker’s management.

**2. Docker Image vs. Docker Containers**

1. **Docker Image**
   * **Layers**: Each instruction in a Dockerfile (e.g., RUN, COPY) creates a new layer, forming a stack of read-only layers.
   * **Distribution**: Stored in registries (Docker Hub, private registries). Can be pulled and run anywhere Docker is installed.
2. **Docker Container**
   * **Writable Layer**: Sits atop the read-only image layers. Changes persist only as long as the container exists.
   * **Ephemeral Nature**: Once removed, any changes in the writable layer are lost unless persisted in volumes.

**3. Docker Image Creation Using Docker Commit & Dockerfile**

1. **docker commit**
   * **Workflow**:
     + Run a container interactively and make changes.
     + Commit those changes to form a new image layer.
   * **Pros/Cons**:
     + **Pro**: Quick and easy for experimentation.
     + **Con**: Not reproducible if you forget which changes were made manually.
2. **Dockerfile**
   * **Structure**:
     + **FROM**: Base image (e.g., FROM ubuntu:20.04).
     + **RUN**: Commands to install software, update packages, etc.
     + **COPY/ADD**: Transfer files from host to container.
     + **CMD/ENTRYPOINT**: Default command or entrypoint for the container.
   * **Benefits**:
     + Fully reproducible and documented build process.
     + Can be tracked in version control for collaborative development.

**4. Dockerfile Important Keywords**

1. **FROM**
   * **Purpose**: Sets the base layer for the new image.
   * **Best Practice**: Use minimal or official images to reduce size and improve security.
2. **RUN**
   * **Function**: Executes commands in a new layer, committing the result.
   * **Examples**: RUN apt-get update && apt-get install -y curl.
3. **COPY / ADD**
   * **Difference**:
     + **COPY**: Copies files/directories from host to container.
     + **ADD**: Similar to COPY but can also handle remote URLs and auto-extract tar archives.
4. **CMD / ENTRYPOINT**
   * **CMD**: Default command that can be overridden at runtime.
   * **ENTRYPOINT**: Defines a fixed executable; additional arguments become parameters.
5. **ENV**
   * **Usage**: Set environment variables in the container (e.g., ENV PORT=8080).
6. **EXPOSE**
   * **Documentation**: Informs about the port the container listens on.
   * **Note**: Does not automatically publish the port on the host.

**5. Docker Tags**

1. **Purpose**
   * Tags differentiate multiple versions of the same image (e.g., myapp:v1.0, myapp:v1.1).
   * latest is a common default but not always indicative of a stable release.
2. **Best Practices**
   * Use semantic versioning (MAJOR.MINOR.PATCH) for clarity.
   * Update documentation to reflect new tags and changes.

**6. Persistent Storage Use-Case**

1. **Why Persistence Matters**
   * Containers are ephemeral, so data stored inside them is lost when they stop.
   * Stateful applications (databases, content management systems) need reliable storage.
2. **Approaches**
   * **Bind Mounts**:
     + Map a host directory to the container.
     + Good for local development, but can be less portable.
   * **Volumes**:
     + Managed by Docker.
     + Easier to back up or migrate, works well in production.
   * **Remote Storage**:
     + NFS, GlusterFS, Amazon EFS, or other network-attached storage solutions.
3. **Data Backup & Recovery**
   * **Regular Snapshots**: Periodically snapshot volumes to guard against data loss.
   * **Testing Restores**: Validate that backups can be restored successfully.

**7. Docker Volumes**

1. **Types**
   * **Named Volumes**: Created and managed by Docker (docker volume create <name>).
   * **Anonymous Volumes**: Created automatically if you don’t specify a name.
   * **Host Mounts**: Directly maps host paths into containers.
2. **Benefits**
   * Decouples data from container lifecycle.
   * Facilitates data sharing between containers (e.g., multiple containers reading/writing logs).
3. **Management**
   * docker volume ls to list volumes.
   * docker volume inspect <name> to see details (mount point, usage).

**8. Docker Networks**

1. **Overview**
   * Containers can communicate over virtual networks created by Docker.
   * By default, Docker sets up a bridge network on a single host.
2. **Network Drivers**
   * **bridge**: Default for standalone containers on a single host.
   * **host**: Container uses the host’s network stack (no isolation).
   * **overlay**: Multi-host networking for Docker Swarm or other orchestration.
   * **macvlan**: Assigns a unique MAC address to each container, making it appear as a separate device on the network.
3. **DNS & Service Discovery**
   * Containers on the same user-defined network can resolve each other by container name.
   * For multi-host setups, orchestration tools often handle DNS-based discovery.

**9. Creating Custom Networks in Docker**

1. **Command Example**
   * docker network create --driver bridge my\_bridge\_network
   * Attach containers using --network my\_bridge\_network when running them.
2. **Benefits of User-Defined Networks**
   * Containers can communicate using automatic DNS resolution (by service/container name).
   * Isolated from the default bridge network, improving security and organization.

**10. Docker Registry**

1. **Definition**
   * A central repository to store and distribute Docker images.
   * Examples: Docker Hub (public), GitHub Container Registry, AWS ECR, Google Container Registry.
2. **Workflow**
   * **Build**: Create the image from a Dockerfile.
   * **Tag**: docker tag local-image registry.example.com/repo/image:version.
   * **Push**: docker push registry.example.com/repo/image:version.
   * **Pull**: docker pull registry.example.com/repo/image:version on any host.
3. **Private Registries**
   * **Security**: Require authentication tokens or credentials.
   * **Use Cases**: Proprietary software, internal images not meant for public consumption.

**11. Docker Inbuilt Security Concepts (Namespaces, cgroups)**

1. **Namespaces**
   * **Purpose**: Provide isolated views of system resources (process IDs, network interfaces, etc.).
   * **Key Namespaces**: pid, net, ipc, mnt, uts, user.
   * **Effect**: A container’s processes appear to run on a separate system, though they share the host kernel.
2. **cgroups (Control Groups)**
   * **Purpose**: Limit and monitor container resource usage (CPU, memory, I/O).
   * **Configuration**: Docker automatically places containers in cgroups. Admins can fine-tune resource limits via docker run flags (--memory, --cpus).
3. **Capabilities**
   * **Linux Capabilities**: Fine-grained privileges (e.g., CAP\_NET\_ADMIN, CAP\_SYS\_TIME).
   * **Default Docker Behavior**: Drops most capabilities to reduce attack surface. You can add them back if needed (--cap-add).
4. **Security Best Practices**
   * Use minimal base images (e.g., Alpine) to reduce vulnerability footprint.
   * Regularly scan images for known CVEs using tools like Trivy or Clair.
   * Avoid running containers as root unless absolutely necessary.

**UNIT VI: Orchestration Tools**

**1. What is Orchestration?**

1. **Definition**
   * Automating the deployment, scaling, networking, and lifecycle management of containerized applications across clusters of machines.
2. **Key Functions**
   * **Scheduling**: Deciding which host runs each container.
   * **Scaling**: Adjusting the number of container instances based on demand.
   * **Load Balancing**: Distributing traffic across containers to prevent overload.
   * **Self-Healing**: Restarting containers that crash or relocating them if a node fails.
3. **Why It Matters**
   * Simplifies operations for complex microservice architectures.
   * Ensures high availability and reliability at scale.

**2. Need of Orchestration**

1. **Complexity with Microservices**
   * Multiple containers and services require coordination for deployment, updates, and discovery.
   * Manual management is error-prone and not scalable.
2. **Scalability & Elasticity**
   * Orchestrators can automatically scale containers based on metrics like CPU usage, memory usage, or custom metrics.
3. **Resilience**
   * If a node in the cluster fails, the orchestrator automatically reschedules containers on healthy nodes.
4. **Deployment Automation**
   * Rolling updates, blue-green deployments, and canary releases can be managed centrally.

**3. Case Study: Need of Orchestration**

1. **Scenario**
   * A large e-commerce site with many microservices (product catalog, cart, checkout, payment gateway).
   * Frequent feature releases and heavy traffic spikes during sales.
2. **Challenges**
   * Manually deploying containers on different servers is time-consuming and error-prone.
   * Managing updates for each microservice without downtime is complex.
3. **Benefits of Orchestration**
   * Automated rollouts/rollbacks reduce risk.
   * Intelligent scheduling optimizes resource usage.
   * Service discovery ensures microservices can find each other reliably.

**4. Container and Microservices**

1. **Microservices Architecture**
   * Applications are broken into small, independently deployable services.
   * Each service focuses on a single business capability and can be scaled independently.
2. **Containers as a Natural Fit**
   * **Isolation**: Each microservice runs in its own container with minimal interference.
   * **Portability**: The same container image can be used in development, testing, and production.
3. **Key Considerations**
   * **Networking**: Services need to communicate with each other, often via REST, gRPC, or message queues.
   * **Data Management**: Each microservice may have its own database or shared storage, depending on the architecture.

**5. Docker Swarm and Kubernetes**

1. **Docker Swarm**
   * **Native Docker Orchestration**: Uses Docker CLI for cluster management.
   * **Features**: Service discovery, load balancing, overlay networking.
   * **Ease of Use**: Simpler to set up than Kubernetes but less feature-rich for complex use cases.
2. **Kubernetes (K8s)**
   * **Industry Standard**: Large ecosystem, extensive community support.
   * **Advanced Features**: Autoscaling (Horizontal Pod Autoscaler), custom resource definitions (CRDs), network policies, etc.
   * **Complexity**: Steeper learning curve, but very powerful and flexible.
3. **Choosing Between Them**
   * **Docker Swarm**: Best for smaller teams/projects that need quick setup.
   * **Kubernetes**: Ideal for large-scale, complex microservices environments with robust operational needs.

**6. Architecture: AWS (ECS/EKS)**

1. **Amazon ECS (Elastic Container Service)**
   * **Orchestrator**: AWS-native solution for running containers on EC2 or AWS Fargate.
   * **Key Concepts**:
     + **Task Definitions**: Specify container image, CPU/memory requirements, and networking.
     + **Services**: Maintain desired number of tasks, integrate with load balancers.
   * **Integration**: Works seamlessly with other AWS services like CloudWatch, IAM, and Route 53.
2. **Amazon EKS (Elastic Kubernetes Service)**
   * **Managed Kubernetes**: AWS handles control plane deployment, upgrades, and high availability.
   * **Worker Nodes**: Can be EC2 or Fargate-based.
   * **EKS Add-ons**: AWS-provided solutions for logging, networking, and security that integrate with K8s.
3. **When to Use ECS vs. EKS**
   * **ECS**: For teams fully aligned with AWS services, wanting less complexity than Kubernetes.
   * **EKS**: For those needing a pure Kubernetes experience or wanting to migrate existing K8s workloads to AWS.

**7. AWS Elastic Container Service Architecture**

1. **Clusters**
   * Logical grouping of container instances or Fargate tasks.
   * You can have multiple clusters for different environments (dev, staging, production).
2. **Task & Service**
   * **Task Definition**: JSON file specifying container parameters (image, ports, environment variables).
   * **Service**: Ensures a specified number of tasks are always running, optionally behind a load balancer.
3. **Networking Modes**
   * **Bridge**: Classic Docker networking.
   * **Host**: Containers share the EC2 instance’s network stack.
   * **awsvpc**: Each task gets its own Elastic Network Interface (ENI) in a VPC, enabling granular security controls.
4. **Scaling**
   * **Service Auto Scaling**: Adjust the number of running tasks based on CPU or memory usage (or custom CloudWatch metrics).
   * **Cluster Auto Scaling**: Dynamically add or remove EC2 instances to accommodate tasks.

**8. Azure Kubernetes Service (AKS)**

1. **Overview**
   * Microsoft’s managed Kubernetes service.
   * Simplifies cluster creation, scaling, and maintenance.
2. **Key Features**
   * **Integration with Azure AD**: For secure access and role-based access control (RBAC).
   * **Azure Monitor**: Unified solution for logs and metrics, integrated with AKS.
   * **Autoscaling**: Node Autoscaling and Pod Autoscaling to match workload demands.
3. **Deployment Flow**
   * Develop and containerize apps locally or in Azure DevOps.
   * Push images to Azure Container Registry (ACR).
   * Deploy to AKS via YAML manifests or Helm charts.

**9. OpenShift**

1. **Definition**
   * Red Hat’s enterprise container platform built on Kubernetes.
   * Offers additional developer-friendly workflows and security features.
2. **Key Features**
   * **Source-to-Image (S2I)**: Automatically build container images from source code without writing Dockerfiles.
   * **Operator Framework**: Streamlines application lifecycle management.
   * **Security Model**: Uses SELinux for strong multi-tenant isolation.
3. **Deployment Models**
   * **OpenShift Container Platform**: On-premises, self-managed.
   * **OpenShift Dedicated**: Hosted by Red Hat, managed for you.
   * **ROSA (Red Hat OpenShift on AWS)**: Jointly managed by Red Hat and AWS.

**10. Kubernetes on Cloud**

1. **General Concept**
   * Running Kubernetes clusters in public clouds (AWS, Azure, GCP) reduces the operational overhead of managing control planes and underlying infrastructure.
2. **Managed Kubernetes Services**
   * **AWS EKS**, **Azure AKS**, **Google GKE**.
   * Cloud providers handle the Kubernetes control plane (API server, etcd), upgrades, and patches.
3. **Benefits**
   * **Auto-Scaling**: Dynamically scale nodes and pods based on usage.
   * **Reduced Ops Burden**: No need to manually manage Kubernetes masters or etcd.
   * **Global Reach**: Deploy clusters in multiple regions for disaster recovery or low latency.
4. **Challenges**
   * **Vendor Lock-In**: Using proprietary cloud features might limit portability.
   * **Cost Management**: Need to monitor resource usage and scale efficiently to avoid large bills.

**11. Monitoring of Container & How to Monitor**

1. **Importance of Monitoring**
   * Containers can be short-lived, so traditional host-based monitoring may not be sufficient.
   * Visibility into performance, resource usage, and application logs is crucial for reliability.
2. **Key Metrics & Telemetry**
   * **CPU/Memory Usage**: Identify resource bottlenecks or memory leaks.
   * **Disk I/O & Network Throughput**: Detect slow database writes or network congestion.
   * **Application Metrics**: Error rates, request latency, throughput, etc.
3. **Popular Monitoring Tools**
   * **Prometheus & Grafana**:
     + Prometheus scrapes metrics from instrumented services.
     + Grafana visualizes data in dashboards.
   * **ELK/EFK Stack**:
     + Elasticsearch for indexing logs.
     + Logstash/Fluentd for log ingestion.
     + Kibana for log analytics.
   * **Commercial Solutions**: Datadog, New Relic, Dynatrace for more comprehensive or managed monitoring.
4. **Alerting & Incident Response**
   * **Alert Rules**: Set thresholds for CPU usage, error rates, or custom metrics.
   * **Escalation Policies**: Determine how alerts are routed (PagerDuty, Opsgenie) and who is on call.
   * **Postmortems**: After an incident, conduct a root cause analysis to improve future responses.
5. **Best Practices**
   * **Instrument Your Code**: Use libraries (OpenTelemetry, StatsD, Prometheus clients) to expose custom metrics.
   * **Centralize Logs**: Don’t rely on container-local logs; forward them to a central system.
   * **Automate**: Ensure that new containers or services automatically register themselves with the monitoring and logging stack.