1. **Develop Azure Compute Solutions**  
    └─ **1.1 Implement Containerized Solutions**  
      └─ **1.1.1 Create and Manage Container Images for Solutions**

- What is the correct structure of a Dockerfile?  
- How do you optimize a Dockerfile for size and performance?  
- How do multi-stage builds work?

- What commands are used to build an image locally?

- How do you tag versions appropriately?

- What is the correct format for image names (registry/repository:tag)?

- How do tags work and how are they used in CI/CD pipelines?

- How do you create and configure an ACR?  
  - What are the SKU tiers and when do you use them?  
  - How do you push/pull images using Docker CLI?  
  - How do you authenticate to ACR (admin account, service principal, managed identity)?  
  - What is az acr login and when is it required?

- How does Azure App Service, Azure Kubernetes Service (AKS), and Azure Container Instances (ACI) consume images from ACR?  
- What are the permission requirements for pulling from ACR?

- How do you scan images for vulnerabilities?  
- What tools or services are used to harden container images?

- How do you manage image versions across environments?  
- How do you clean up unreferenced or old images in ACR?

- How can ACR tasks automate image builds?  
- How do you use az acr task to create scheduled or event-driven builds?

- What are the pros and cons of storing images publicly vs privately?  
- How do you configure access control for image repositories?

**What is the correct structure of a Dockerfile?**

FROM <base\_image>

[ENV <key>=<value> ...]

[WORKDIR <directory>]

[COPY <src> <dest>]

[RUN <command>]

[EXPOSE <port>]

[CMD ["executable", "param1", ...]]

**Key Points:**

* FROM must be first (defines base image).
* WORKDIR sets working directory inside container.
* COPY adds files to the image from build context.
* RUN executes shell commands (e.g., install packages).
* EXPOSE is optional metadata; doesn't actually open ports.
* CMD defines default container startup command (only one allowed; last one wins).

**How do you optimize a Dockerfile for size and performance?**

1. **Use slim/minimal base images:**  
    Prefer FROM mcr.microsoft.com/dotnet/aspnet:7.0-alpine over full images.
2. **Leverage multi-stage builds:**  
    Build in one stage, copy only final output to runtime stage to reduce size.
3. **Minimize layers:**  
    Group related RUN commands and clean up temp files in the same layer:
4. **Avoid unnecessary files:**  
    Use .dockerignore to exclude files (e.g., .git, node\_modules).
5. **Set only needed environment variables and permissions:**  
    Avoid excessive ENV or USER changes unless required.

**How do multi-stage builds work?**

Reduce final image size by separating build and runtime stages.

* Define multiple FROM statements in one Dockerfile.
* Use an alias for the build stage (AS build).
* Copy only needed artifacts from the build stage into the final image.

**What commands are used to build an image locally?**

docker build -t <name>:<tag> <path>

e.g. *docker build -t myapp:latest .*

**How do you tag versions appropriately?**

Use semantic tags like:  
**latest, v1.0.0, dev, staging**

e.g. *docker build -t myapp:v1.0.0 .*

Tag meaning should reflect version or environment for clarity and traceability.

**What is the correct format for image names?**

*<registry>/<repository>:<tag> :  
myregistry.azurecr.io/myapp:v1.0.0*

Registry is optional for local images. Tag defaults to **latest** if omitted.

**How do tags work and how are they used in CI/CD pipelines?**

Tags identify image versions. Pipelines use tags to pull, test, and deploy specific builds:  
**docker push myapp:staging** → used in staging environment.  
**latest** often used in dev, **versioned tags** in prod.

**How do you create and configure an ACR?**

* Create ACR: **az acr create --name <acr-name> --resource-group <rg> --sku Basic**
* Enable admin access: **az acr update -n <acr-name> --admin-enabled true**
* Login: **az acr login --name <acr-name>**

**What are the SKU tiers and when do you use them?**

* **Basic** – Dev/test, low-cost, limited features
* **Standard** – Prod-ready, geo-replication support
* **Premium** – High-scale, content trust, private endpoints, more throughput

**How do you push/pull images from/to ACR using Docker CLI?**

**Push:**

1. **docker tag myapp myacr.azurecr.io/myapp**
2. **docker push myacr.azurecr.io/myapp**

**Pull:**

**docker pull myacr.azurecr.io/myapp**

**How do you authenticate to ACR (admin, service principal, managed identity)?**

* **Admin account:**  
  Enable with **az acr update --admin-enabled true**, then use provided username/password.
* **Service principal:**  
  Assign AcrPush/AcrPull role, login with docker login using SP credentials.
* **Managed identity:**  
  Grant role to identity, Azure services (e.g., App Service) authenticate automatically.

**What is az acr login and when is it required?**

*az acr login --name <acr-name>*  
Authenticates Docker CLI with ACR.  
Required for manual Docker pushes/pulls. Not needed for Azure services using managed identity.

**How does Azure App Service, Azure Kubernetes Service (AKS), and Azure Container Instances (ACI) consume images from ACR?**

* **App Service:** Configure container settings with ACR URL; use managed identity or admin acc.
* **AKS:** Enable ACR integration via az aks update or use imagePullSecrets.
* **ACI:** Reference image with full ACR path; grant access via identity or admin credentials.

**What are the permission requirements for pulling from ACR?**

The identity must have **AcrPull** role on the ACR.  
Can be assigned to:

* User
* Service principal
* Managed identity (App Service, AKS, etc.)

**How do you scan images for vulnerabilities?**

Use **Microsoft Defender for Cloud** with ACR integration. It scans images on push and shows CVEs in the portal. Enable under *Defender plans > Container registries*.

**What tools or services are used to harden container images?**

* **Microsoft Defender for Cloud** – vulnerability scanning
* **Dockerfile best practices** – minimize layers, use minimal base images
* **Content trust** – ensure image integrity
* **Private ACR** – restrict access
* **ACR Tasks** – automate secure builds

**How do you manage image versions across environments?**

Use consistent **tagging strategy** (e.g., dev, staging, v1.0.0).  
Promote images by **re-tagging** and pushing to ACR for each stage.

**How do you clean up unreferenced or old images in ACR?**

Use **ACR Tasks with retention policies** or manual cleanup via:  
**az acr repository delete --name <acr> --image <repo>:<tag>**

**How can ACR tasks automate image builds?**

**ACR Tasks** can auto-build images on source code or base image changes:  
**az acr task create** with --source and --cmd "docker build"

Supports triggers (e.g., Git push) and scheduling.

**How do you use az acr task to create scheduled or event-driven builds?**

**Event-driven:**

az acr task create \

--name mytask \

--registry myacr \

--image myapp:{{.Run.ID}} \

--context https://github.com/org/repo.git \

--file Dockerfile \

**--git-access-token <token>**

**Scheduled:**

az acr task create \

--name mytask \

--registry myacr \

**--schedule "0 2 \* \* \*" \**

--image myapp:nightly \

--context https://github.com/org/repo.git \

--file Dockerfile

**What are the pros and cons of storing images publicly vs privately?**

**Public:** Easy access, no auth needed — but insecure, no access control.  
**Private:** Secure, controlled access — but needs auth, may cost more.

**How do you configure access control for image repositories?**

Assign **AcrPull** or **AcrPush** roles to users, service principals, or managed identities using **Azure RBAC** on the ACR resource.