Admissions & Support:
support@mockattempt.com
Phone / WhatsApp: +91 95400 07917
Course Code: DOCK • **Updated:** 05 Nov 2026
https://mockattempt.com/courses/docker-training

DOCK • CONTAINERS, BUILDKIT & CLOUD-NATIVE PACKAGING | **BEGINNER TO SENIOR DOCKER & CONTAINER SECURITY ENGINEER**

100% LIVE INDUSTRY EXPERT COHORT + INTERNSHIP LETTER

Docker Training

Master Docker from Linux Kernel Namespaces, cgroups v2, containerd/runc & WSL2 Internals to BuildKit Cache Mounts, Distroless Multi-Stage Dockerfiles (Python, Java, PHP, Ruby, Scala), Overlay2 Storage, Docker Compose v2 Healthchecks & OCI Registry Signing — with 200 Mock Tests & Voice MockAttempt AI Interviews at ₹1,999 (Incl. 18% GST).

Duration: 14–16 Weeks • Live + Hands-On Projects (205 Hrs) • **Modules:** 16 • **Live Topics:** 42 • **Subtopics:** 336+ • **Labs & Tasks:** 84 • **Mocks / AI Vivas:** 244 / 244 • **Projects:** 16 Module + 1 Live Client Prod

All-Inclusive Direct Fee (Incl. 18% GST • Zero Hidden Charges): Total Payable: ₹1,999 (Net Tuition ₹1,694 + Included 18% GST ₹305) • **Split Option:** 100% Credit to Job Track • **Effective ROI:** ~₹10/Guided Hr (~₹48/Live Topic)

Faculty: 100% Active Working IT Professionals • **Live Batch Tracks (7:00 AM – 8:00 PM IST Flexible Slots):** Weekday Live Online Batch (Mon – Fri • 1 Hr/Day • 7:00 AM – 8:00 PM) (Monday to Friday (1 Hour Live Session / Day) • 07:00 AM – 08:00 PM IST (Students Can Choose Any 1-Hour Slot Based on Trainer Availability) • 17–19 Weeks Fast-Track) | Weekend Live Online Batch (Sat – Sun • 2 Hrs/Day • 7:00 AM – 8:00 PM) (Saturday & Sunday (2 Hours Live Session / Day) • 07:00 AM – 08:00 PM IST (Students Can Choose Any 2-Hour Slot Based on Trainer Availability) • 19–21 Weeks Weekend Track)

Core Technologies & Engineering Stack: Docker Engine 27 & containerd / runc • BuildKit & Multi-Stage Dockerfiles • Linux Namespaces, cgroups v2 & Seccomp • Overlay2 Copy-on-Write Storage • Docker Compose v2 & Bridge Networks • Distroless / Alpine Non-Root Images • OCI Registries, Trivy & Cosign

1. PROGRAM MASTERY OUTCOMES & ELIGIBILITY

End-to-End Engineering Outcomes:

1. Explain how Linux Kernel Namespaces (`pid`, `net`, `mnt`, `uts`, `ipc`, `user`) and `cgroups v2` isolate containers without guest hypervisor OS overhead
2. Author tiny, vulnerability-minimal Multi-Stage `Dockerfile`s across Python, Java 21, PHP-FPM, Ruby, and Scala using BuildKit cache/secret mounts and non-root `USER`s
3. Orchestrate multi-service local and staging stacks using Docker Compose v2 (`service_healthy` dependencies, named volumes, isolated bridge networks) and push multi-arch OCI images
4. Clear 244 Timed Mock Exams (10,000 Docker & Container Security MCQs) and 200 Spoken Voice + MockAttempt AI Container Engineering Interviews
5. Linux namespaces, cgroups, OverlayFS layer mechanics, and deterministic `.dockerignore` build contexts.
6. Minimal Multi-Stage Dockerfiles (`alpine`/`distroless`), build-cache ordering, BuildKit secrets, and non-root `USER` execution.
7. Multi-container Docker Compose stacks: custom bridge networks, named volumes, healthcheck `depends_on`, and resource limits.
8. Container supply-chain security: vulnerability scanning, read-only root filesystems, and private registry tagging.

Eligibility & Prerequisites: Basic terminal/shell navigation — starts from container vs virtual machine fundamentals from scratch • Any Windows 10/11 (WSL 2), macOS, or Linux laptop/PC capable of running Docker Engine or Docker Desktop • No prior Docker industry experience required — Module 1 starts from environment setup, core mental models, and plain-English fundamentals. • Ideal for B.Tech / BCA / MCA / B.Sc IT students, non-CS beginners, career switchers, and working developers upgrading to 2026 production standards. • Commitment to attend live Weekday or Weekend classes, complete daily hands-on lab checklists, and practice topic-wise Mock Tests & AI Interviews.

2. DETAILED ZERO-TO-EXPERT MODULE, TOPIC & SUBTOPIC SYLLABUS (16 MODULES • 42 LIVE TOPICS • 336+ SUBTOPICS)

Progressive Zero-to-Expert Mastery Stages: Modules 1–4: Stage 1: Day-1 Foundations & Core Mental Models (8 Topics / 64 Subtopics) • Modules 5–8: Stage 2: Applied Engineering, Data & API Workflows (8 Topics / 64 Subtopics) • Modules 9–12: Stage 3: Internals, Concurrency, Security & Performance (12 Topics / 96 Subtopics) • Modules 13–16: Stage 4: Production Architecture, Testing, CI/CD & Expert Mastery (14 Topics / 112 Subtopics)

MODULE 1 OF 16 • STAGE 1 • 11 LIVE + LAB HOURS • 2 TOPICS (16 SUBTOPICS) • 1 CAPSTONE PROJECT (100 MARKS)

Module 1: OS-Level Virtualization, Linux Kernel Primitives & Core Docker Features

Understand why containers boot in milliseconds by sharing the host Linux kernel instead of booting heavyweight guest OS hypervisors, and explore OCI standardization, immutability, and cgroups v2 resource controls.

Objectives: ✓ Docker Fundamentals — Containers vs Virtual Machines, Shared Host Kernel & OCI Standards — Why VMs duplicate full guest kernels and gigabytes of OS RAM per instance while Docker containers run as isolated user-space Linux processes sharing the host kernel, governed by Open Container Initiative (`OCI` image-spec & runtime-spec) standards. • ✓ Docker Core Features — Linux Namespaces, `cgroups v2`, Seccomp/Capabilities & Layer Deduplication — How Docker achieves isolation + security + speed: Namespaces isolate **visibility** (`pid`, `net`, `mnt`, `uts`, `ipc`, `user`), `cgroups v2` meter **hardware consumption** (CPU, RAM, Block I/O, PIDs), Linux Capabilities + `seccomp` filter ~44 risky syscalls by default, and read-only layers deduplicate disk/network pulls. • ✓ Build, verify, and ship Module Capstone: Zero-Drift Immutable Microservice Artifact & Kernel Resource Boundary Spec.

1.1 Docker Fundamentals — Containers vs Virtual Machines, Shared Host Kernel & OCI Standards

[50 mins • Assignment (100M) • Hands-On Lab (50M) • 10 Mock Sets • 10 AI Vivas]

Subtopics Covered: Shared host Linux kernel processes & Hypervisor VM vs Container density • Core Syntax, CLI & Production APIs: docker version, docker info, OCI runtime-spec (runc) & OCI image-spec • Deep-Dive Mechanics: OCI image-spec & runtime-spec & Ephemeral disposable container state • Advanced Patterns: PID 1 process lifecycle & signals & Content-addressable SHA-256 digests • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab: Minimal Reproducible OCI Container Image Spec (`Dockerfile`) • Idiomatic DOCK Implementation: multi-stage BuildKit Dockerfiles, distroless/Alpine images & Compose networks • Linux Namespaces, cgroups v2, OverlayFS Layers & containerd/runc OCI Execution & Memory Lifecycle for Docker Fundamentals

Purpose: To understand and apply Docker Fundamentals — Containers vs Virtual Machines, Shared Host Kernel & OCI Standards in real Docker systems — specifically: Why VMs duplicate...

What Topic Is Next → Next Topic 1.2: Docker Core Features — Linux Namespaces, `cgroups v2`, Seccomp/Capabilities & Layer Deduplication

1.2 Docker Core Features — Linux Namespaces, `cgroups v2`, Seccomp/Capabilities & Layer Deduplication

[50 mins • Assignment (100M) • Hands-On Lab (50M) • 10 Mock Sets • 10 AI Vivas]

Subtopics Covered: Namespaces (visibility) vs cgroups (limits) & Default seccomp ~44 syscall blocklist • Core Syntax, CLI & Production APIs: --cpus, --memory, --pids-limit, --cap-drop=ALL, --security-opt=no-new-privileges:true • Deep-Dive Mechanics: Linux Capabilities (--cap-drop=ALL) & Never use --privileged in production • Advanced Patterns: No-new-privileges setuid guard & SHA-256 read-only layer deduplication • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab: Hardened Capability-Minimal `compose.yaml` Service Spec • Idiomatic DOCK Implementation: multi-stage BuildKit Dockerfiles, distroless/Alpine images & Compose networks • Linux Namespaces, cgroups v2, OverlayFS Layers & containerd/runc OCI Execution & Memory Lifecycle for Docker Core Features

Purpose: To understand and apply Docker Core Features — Linux Namespaces, `cgroups v2`, Seccomp/Capabilities & Layer Deduplication in real Docker systems — specifically: How Docke...

What Topic Is Next → Module 1 Capstone (Zero-Drift Immutable Microservice Artifact & Kernel Resource Boundary Spec) → Next Topic 2.1: Docker Architecture — Client REST API, `dockerd`, `containerd`, `containerd-shim` & `runc`

Module 1 Hands-On Capstone Project (100 Marks): Zero-Drift Immutable Microservice Artifact & Kernel Resource Boundary Spec — Package a stateless HTTP health probe into a minimal non-root OCI image pinned by SHA-256 digest and verify strict CPU/memory cgroup boundaries at runtime.

MODULE 2 OF 16 • STAGE 1 • 11 LIVE + LAB HOURS • 2 TOPICS (16 SUBTOPICS) • 1 CAPSTONE PROJECT (100 MARKS)

Module 2: Docker Engine Daemon Architecture (`dockerd` / `containerd` / `runc`) & Linux Installation

Trace every `docker` CLI REST call through `dockerd`, `containerd`, `containerd-shim`, and `runc`, and harden Linux Engine installations with `daemon.json` log rotation and rootless security.

Objectives: ✓ Docker Architecture — Client REST API, `dockerd`, `containerd`, `containerd-shim` & `runc` — Why modern Docker split the monolithic daemon into modular layers: `docker` CLI speaks HTTP over `/var/run/docker.sock` to `dockerd`, which delegates lifecycle over gRPC to `containerd`; `containerd` spawns a per-container `containerd-shim` that invokes OCI `runc` to create kernel namespaces and then exits — leaving the shim as lightweight parent!. • ✓ Installing Docker Engine on Linux — GPG Apt Repos, Rootless Mode & `docker` Group Security — Removing conflicting legacy distro packages, adding Docker's official `/etc/apt/keyrings/docker.asc` GPG repository (`docker-ce`, `docker-ce-cli`, `containerd.io`, `docker-buildx-plugin`, `docker-compose-plugin`), understanding `usermod -aG docker` root-equivalence, and running Rootless Docker via `user_namespaces`. • ✓ Build, verify, and ship Module Capstone: Production `/etc/docker/daemon.json` Hardening & Disk-Exhaustion Guard.

2.1 Docker Architecture — Client REST API, `dockerd`, `containerd`, `containerd-shim` & `runc`

[50 mins • Assignment (100M) • Hands-On Lab (50M) • 10 Mock Sets • 10 AI Vivas]

Subtopics Covered: CLI -> docker.sock -> dockerd REST & Dockerd -> containerd gRPC delegation • Core Syntax, CLI & Production APIs: /var/run/docker.sock, dockerd, containerd, containerd-shim-runc-v2, runc, live-restore • Deep-Dive Mechanics: Per-container containerd-shim parent & Runc OCI namespace/cgroup creator • Advanced Patterns: Docker.sock = root host equivalence & Live-restore zero-downtime daemon restart • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab: Production-Hardened `/etc/docker/daemon.json` Configuration • Idiomatic DOCK Implementation: multi-stage BuildKit Dockerfiles, distroless/Alpine images & Compose networks • Linux Namespaces, cgroups v2, OverlayFS Layers & containerd/runc OCI Execution & Memory Lifecycle for Docker Architecture

Purpose: To understand and apply Docker Architecture — Client REST API, `dockerd`, `containerd`, `containerd-shim` & `runc` in real Docker systems — specifically: Why modern Docke...

What Topic Is Next → Next Topic 2.2: Installing Docker Engine on Linux — GPG Apt Repos, Rootless Mode & `docker` Group Security

2.2 Installing Docker Engine on Linux — GPG Apt Repos, Rootless Mode & `docker` Group Security

[50 mins • Assignment (100M) • Hands-On Lab (50M) • 10 Mock Sets • 10 AI Vivas]

Subtopics Covered: Official GPG keyring apt repository & Docker-ce + containerd.io + buildx • Core Syntax, CLI & Production APIs: docker-ce, containerd.io, docker-buildx-plugin, docker-compose-plugin, dockerd-rootless-setu... • Deep-Dive Mechanics: Docker compose v2 Go plugin (space) & Docker group = host root equivalence • Advanced Patterns: Rootless Docker userns remapping & Mandatory daemon.json log rotation • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab: Automated Ubuntu 24.04 LTS Docker Engine Bootstrap Verification Dockerfile • Idiomatic DOCK Implementation: multi-stage BuildKit Dockerfiles, distroless/Alpine images & Compose networks • Linux Namespaces, cgroups v2, OverlayFS Layers & containerd/runc OCI Execution & Memory Lifecycle for Installing Docker Engine on

Linux

Purpose: To understand and apply Installing Docker Engine on Linux — GPG Apt Repos, Rootless Mode & `docker` Group Security in real Docker systems — specifically: Removing conflic...

What Topic Is Next → Module 2 Capstone (Production `/etc/docker/daemon.json` Hardening & Disk-Exhaustion Guard) → Next Topic 3.1: Docker on Windows 10/11 — WSL 2 Lightweight Linux Kernel, `.wslconfig` & Avoiding `/mnt/c` I/O Traps

Module 2 Hands-On Capstone Project (100 Marks): Production `/etc/docker/daemon.json` Hardening & Disk-Exhaustion Guard — Configure a production Linux Docker Engine daemon spec that prevents unbounded JSON container logs from filling `/var/lib/docker`, enables `live-restore`, and enforces BuildKit + `overlay2`.

MODULE 3 OF 16 • STAGE 1 • 11 LIVE + LAB HOURS • 2 TOPICS (16 SUBTOPICS) • 1 CAPSTONE PROJECT (100 MARKS)

Module 3: Docker on Windows (`WSL 2` Architecture) & Image vs Container Layer Mechanics

Optimize Docker Desktop on Windows 11 with WSL 2 native `ext4` filesystems (avoiding `/mnt/c` 9P bottlenecks), and master read-only image layers vs the thin writable container layer.

Objectives: ✓ Docker on Windows 10/11 — WSL 2 Lightweight Linux Kernel, `.wslconfig` & Avoiding `/mnt/c` I/O Traps — How Docker Desktop on Windows runs a real Microsoft-maintained Linux kernel inside a lightweight WSL 2 utility VM, why bind-mounting C:\ (`/mnt/c/`) crosses a slow 9P protocol bridge (`10x–20x` slower `npm install` / Composer / Maven I/O!), and capping `vmmem` via `.wslconfig`. • ✓ Docker Images vs Containers — Read-Only Content-Addressable Layers vs Thin Writable R/W Layer — Class vs Object analogy at the OS level: an **Image** is an immutable stack of read-only filesystem diff layers + JSON config (entrypoint, env, user), while a **Container** mounts a thin ephemeral Read-Write Copy-on-Write (`upperdir`) layer on top of those shared read-only layers (`lowerdir`). • ✓ Build, verify, and ship Module Capstone: High-Speed WSL 2 Developer Workspace & Multi-Arch Image Manifest Audit.

3.1 Docker on Windows 10/11 — WSL 2 Lightweight Linux Kernel, `.wslconfig` & Avoiding `/mnt/c` I/O Traps [50 mins • Assignment (100M) • Hands-On Lab (50M) • 10 Mock Sets • 10 AI Vivas]

Subtopics Covered: WSL 2 lightweight Linux kernel VM & WSL ext4 vs slow /mnt/c 9P bind mount • Core Syntax, CLI & Production APIs: wsl --install, wsl --status, %USERPROFILE%\.wslconfig, WSL 2 ext4 vs /mnt/c 9P DrvFs • Deep-Dive Mechanics: .wslconfig memory & processors cap & CRLF (\r\n) shebang /bin/sh\r trap • Advanced Patterns: .gitattributes eol=lf enforcement & Windows Containers vs Linux mode • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab: CRLF-Proof Multi-OS Safe Entrypoint Normalization in `Dockerfile` • Idiomatic DOCK Implementation: multi-stage BuildKit Dockerfiles, distroless/Alpine images & Compose networks • Linux Namespaces, cgroups v2, OverlayFS Layers & containerd/runc OCI Execution & Memory Lifecycle for Docker on Windows 10/11

Purpose: To understand and apply Docker on Windows 10/11 — WSL 2 Lightweight Linux Kernel, `.wslconfig` & Avoiding `/mnt/c` I/O Traps in real Docker systems — specifically: How Do...

What Topic Is Next → Next Topic 3.2: Docker Images vs Containers — Read-Only Content-Addressable Layers vs Thin Writable R/W Layer

3.2 Docker Images vs Containers — Read-Only Content-Addressable Layers vs Thin Writable R/W Layer [50 mins • Assignment (100M) • Hands-On Lab (50M) • 10 Mock Sets • 10 AI Vivas]

Subtopics Covered: Immutable Read-Only Image layers & Ephemeral Read-Write Container layer • Core Syntax, CLI & Production APIs: docker images --digests, docker history <img >, docker ps -a -s, docker inspect • Deep-Dive Mechanics: Whiteout files do not shrink prior layers & OCI Image Index multi-arch manifest • Advanced Patterns: Docker ps -s (size vs virtual size) & Tag mutability vs @sha256 immutability • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab: Single-Layer Cleanup Pattern Preventing Bloated Whiteout Layers • Idiomatic DOCK Implementation: multi-stage BuildKit Dockerfiles, distroless/Alpine images & Compose networks • Linux Namespaces, cgroups v2, OverlayFS Layers & containerd/runc OCI Execution & Memory Lifecycle for Docker Images vs Containers

Purpose: To understand and apply Docker Images vs Containers — Read-Only Content-Addressable Layers vs Thin Writable R/W Layer in real Docker systems — specifically: Class vs Obje...

What Topic Is Next → Module 3 Capstone (High-Speed WSL 2 Developer Workspace & Multi-Arch Image Manifest Audit) → Next Topic 4.1: Mastering `docker run` — Lifecycle (`create` + `start`), `127.0.0.1` Port Binding, `--init` & Resource Caps

Module 3 Hands-On Capstone Project (100 Marks): High-Speed WSL 2 Developer Workspace & Multi-Arch Image Manifest Audit — Eliminate 10x Windows bind-mount I/O slowdowns using WSL 2 native `ext4` storage + `.wslconfig` bounds, and inspect multi-architecture (`linux/amd64`, `linux/arm64`) OCI image manifests.

MODULE 4 OF 16 • STAGE 1 • 11 LIVE + LAB HOURS • 2 TOPICS (16 SUBTOPICS) • 1 CAPSTONE PROJECT (100 MARKS)

Module 4: `docker run` Production Flags, Signal Handling (`PID 1`) & BuildKit `Dockerfile` Craft

Run containers safely with explicit loopback port bindings, `--init` zombie reaping, and resource caps, and write cache-optimal `Dockerfile`s (`COPY` vs `ADD`, `CMD` vs `ENTRYPOINT`, `.dockerignore`).

Objectives: ✓ Mastering `docker run` — Lifecycle (`create` + `start`), `127.0.0.1` Port Binding, `--init` & Resource Caps — What `docker run` orchestrates under the hood (pull -> create -> network netns/veth -> mount volumes -> start PID 1), `-d` detached vs `-it` interactive TTY, `--rm` auto-cleanup, `--init` (`tini` zombie reaper), and why `-p 8080:8080` binds `0.0.0.0` bypassing host `ufw` rules!. • ✓ Production `Dockerfile` Engineering — Layer Cache Ordering, `COPY` vs `ADD`, `ARG` vs `ENV` & `ENTRYPOINT` — How BuildKit invalidates every downstream layer after a cache miss (copy dependency lockfiles BEFORE source code!), why `COPY` is preferred over magic-tar/URL `ADD`, `ARG` (build-time) vs `ENV` (persisted runtime), BuildKit `--mount=type=secret` (never put secrets in `ARG`!), and JSON exec `ENTRYPOINT` + `CMD`. • ✓ Build, verify, and ship Module Capstone: Cache-Optimal BuildKit `Dockerfile` & Firewall-Safe Runtime Launch Spec.

4.1 Mastering `docker run` — Lifecycle (`create` + `start`), `127.0.0.1` Port Binding, `--init` & Resource Caps [50 mins • Assignment (100M) • Hands-On Lab (50M) • 10 Mock Sets • 10 AI Vivas]

Subtopics Covered: Pull + create + network + start pipeline & -p 0.0.0.0 UFW bypass vs 127.0.0.1 • Core Syntax, CLI & Production APIs: docker run -d --name <n> --init --restart unless-stopped -p 127.0.0.1:8080:8080 --memory=512... • Deep-Dive Mechanics: --init (tini) PID 1 zombie reaper & --memory & --cpus cgroup hard caps • Advanced Patterns: --restart unless-stopped vs on-failure & --read-only + --tmpfs /tmp runtime • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab:

Firewall-Safe & Resource-Capped `compose.yaml` Equivalent of `docker run` • Idiomatic DOCK Implementation: multi-stage BuildKit Dockerfiles, distroless/Alpine images & Compose networks • Linux Namespaces, cgroups v2, OverlayFS Layers & containerd/runc OCI Execution & Memory Lifecycle for Mastering `docker run`

Purpose: To understand and apply Mastering `docker run` — Lifecycle (`create` + `start`), `127.0.0.1` Port Binding, `--init` & Resource Caps in real Docker systems — specifically:...

What Topic Is Next → Next Topic 4.2: Production `Dockerfile` Engineering — Layer Cache Ordering, `COPY` vs `ADD`, `ARG` vs `ENV` & `ENTRYPOINT`

4.2 Production `Dockerfile` Engineering — Layer Cache Ordering, `COPY` vs `ADD`, `ARG` vs `ENV` & `ENTRYPOINT` [50 mins • Assignment (100M) • Hands-On Lab (50M) • 10 Mock Sets • 10 AI Vivas]

Subtopics Covered: Lockfile-first layer cache ordering & .dockerignore context & secret guard • Core Syntax, CLI & Production APIs: FROM ... AS builder, WORKDIR, COPY --chown, RUN --mount=type=cache, ARG vs ENV, HEALTHCHECK,... • Deep-Dive Mechanics: COPY explicit vs ADD tar auto-extract & ARG (docker history visible!) vs secret mount • Advanced Patterns: Exec-form ["cmd"] SIGTERM delivery & Multi-Stage FROM ... AS builder • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab: BuildKit Secret-Safe & Cache-Mounted Multi-Stage `Dockerfile` • Idiomatic DOCK Implementation: multi-stage BuildKit Dockerfiles, distroless/Alpine images & Compose networks • Linux Namespaces, cgroups v2, OverlayFS Layers & containerd/runc OCI Execution & Memory Lifecycle for Production `Dockerfile` Engineering

Purpose: To understand and apply Production `Dockerfile` Engineering — Layer Cache Ordering, `COPY` vs `ADD`, `ARG` vs `ENV` & `ENTRYPOINT` in real Docker systems — specifically:...

What Topic Is Next → Module 4 Capstone (Cache-Optimal BuildKit `Dockerfile` & Firewall-Safe Runtime Launch Spec) → Next Topic 5.1: Essential Docker CLI Commands — `exec`, `logs`, `stats`, `cp`, `events` & Safe `system prune`

Module 4 Hands-On Capstone Project (100 Marks): Cache-Optimal BuildKit `Dockerfile` & Firewall-Safe Runtime Launch Spec — Refactor a naive slow-rebuilding `Dockerfile` so source-only edits rebuild in `<1s` via layer caching, exclude `.git`/`node_modules`/`.env` via `.dockerignore`, and handle `SIGTERM` cleanly...

MODULE 5 OF 16 • STAGE 2 • 11 LIVE + LAB HOURS • 2 TOPICS (16 SUBTOPICS) • 1 CAPSTONE PROJECT (100 MARKS)

Module 5: Essential Docker Operations CLI & Production Python Containerization

Debug live containers using `docker exec`, `logs`, `stats`, `events`, and `system df/prune`, and package fast, non-root Python (`uv` / Gunicorn / FastAPI) microservices cleanly.

Objectives: ✓ Essential Docker CLI Commands — `exec`, `logs`, `stats`, `cp`, `events` & Safe `system prune` — Daily operational commands: opening diagnostic shells (`docker exec -it <c> sh`), tailing timestamped logs (`docker logs -f --tail 100 -t`), live CPU/Memory/Net I/O (`docker stats --no-stream`), inspecting disk reclamation (`docker system df`), and understanding `docker buildx prune` vs dangerous `docker system prune --volumes`. • ✓ Containerizing Python Applications — `python:3.12-slim`, `PYTHONUNBUFFERED=1` & Multi-Stage `venv` — Why `python:3.12` full image weighs ~1 GB whereas `python:3.12-slim` + multi-stage `/opt/venv` shrinks to ~130 MB, why Alpine `musl` libc breaks binary `manylinux` PyPI wheels (`numpy`, `psycopg`, `cryptography`) forcing slow C compilation, and why `PYTHONUNBUFFERED=1` is mandatory. • ✓ Build, verify, and ship Module Capstone: Multi-Stage Non-Root Python 3.12 Gunicorn/Uvicorn Production Image.

5.1 Essential Docker CLI Commands — `exec`, `logs`, `stats`, `cp`, `events` & Safe `system prune` [50 mins • Assignment (100M) • Hands-On Lab (50M) • 10 Mock Sets • 10 AI Vivas]

Subtopics Covered: ExitCode 137 (OOM/SIGKILL) vs 143 (SIGTERM) & Docker stats --no-stream cgroup telemetry • Core Syntax, CLI & Production APIs: docker exec -it, docker logs --tail -f, docker stats --no-stream, docker system df, docker i... • Deep-Dive Mechanics: Docker system df disk breakdown & Never prune --volumes carelessly • Advanced Patterns: Docker events real-time daemon stream & Docker exec vs docker attach trap • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab: Diagnostic Exit-Code & Health Inspector Container Script • Idiomatic DOCK Implementation: multi-stage BuildKit Dockerfiles, distroless/Alpine images & Compose networks • Linux Namespaces, cgroups v2, OverlayFS Layers & containerd/runc OCI Execution & Memory Lifecycle for Essential Docker CLI Commands

Purpose: To understand and apply Essential Docker CLI Commands — `exec`, `logs`, `stats`, `cp`, `events` & Safe `system prune` in real Docker systems — specifically: Daily operati...

What Topic Is Next → Next Topic 5.2: Containerizing Python Applications — `python:3.12-slim`, `PYTHONUNBUFFERED=1` & Multi-Stage `venv`

5.2 Containerizing Python Applications — `python:3.12-slim`, `PYTHONUNBUFFERED=1` & Multi-Stage `venv` [50 mins • Assignment (100M) • Hands-On Lab (50M) • 10 Mock Sets • 10 AI Vivas]

Subtopics Covered: Python:3.12-slim glibc wheel speed & Alpine musl breaks manylinux wheels • Core Syntax, CLI & Production APIs: FROM python:3.12-slim, ENV PYTHONUNBUFFERED=1 PYTHONDONTWRITEBYTECODE=1, PATH="/opt/venv/bin... • Deep-Dive Mechanics: PYTHONUNBUFFERED=1 real-time logs & PYTHONDONTWRITEBYTECODE=1 clean layers • Advanced Patterns: Multi-Stage /opt/venv PATH pattern & Urllib native HEALTHCHECK (no curl) • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab: Multi-Stage Non-Root Python 3.12 `venv` Production `Dockerfile` • Idiomatic DOCK Implementation: multi-stage BuildKit Dockerfiles, distroless/Alpine images & Compose networks • Linux Namespaces, cgroups v2, OverlayFS Layers & containerd/runc OCI Execution & Memory Lifecycle for Containerizing Python Applications

Purpose: To understand and apply Containerizing Python Applications — `python:3.12-slim`, `PYTHONUNBUFFERED=1` & Multi-Stage `venv` in real Docker systems — specifically: Why `pyt...

What Topic Is Next → Module 5 Capstone (Multi-Stage Non-Root Python 3.12 Gunicorn/Uvicorn Production Image) → Next Topic 6.1: Containerizing Java 21 & Spring Boot — `MaxRAMPercentage`, Non-Heap Headroom & Layered JREs

Module 5 Hands-On Capstone Project (100 Marks): Multi-Stage Non-Root Python 3.12 Gunicorn/Uvicorn Production Image — Containerize a Python 3.12 web API using a two-stage virtual-environment build so the final runtime image contains zero `gcc`/`pip` build toolchains, streams unbuffered JSON logs, and runs a...

MODULE 6 OF 16 • STAGE 2 • 11 LIVE + LAB HOURS • 2 TOPICS (16 SUBTOPICS) • 1 CAPSTONE PROJECT (100 MARKS)

Module 6: Enterprise JVM (`Java 21` `jlink` / Layered JARs) & `PHP-FPM + Nginx` Containers

Containerize Java 21 / Spring Boot 3 services with cgroup-aware `MaxRAMPercentage` and layered JREs, and architect production PHP 8.3 (`php-fpm` + OPcache + Nginx) container topologies.

Objectives: ✓ Containerizing Java 21 & Spring Boot — `MaxRAMPercentage`, Non-Heap Headroom & Layered JREs — Why shipping a full `jdk` adds ~180 MB of unneeded `javac`/`jmap` attack surface vs `eclipse-temurin:21-jre`, why `-Xmx512m` inside a `512Mi` container STILL triggers Linux `OOMKilled (137)` (Heap + Metaspace + Thread stacks + Direct Buffers > 512Mi!), and Spring Boot `jar -Djarmode=tools extract --layers`. • ✓ Containerizing PHP 8.3 & Laravel — Multi-Stage Composer, `php-fpm` Tuning & Immutable `OPcache` — Installing compiled extensions (`pdo_mysql`, `intl`, `redis`, `opcache`), copying `composer:2` binary cleanly (`COPY --from=composer:2 /usr/bin/composer /usr/bin/composer`), `--no-dev --optimize-autoloader --classmap-authoritative`, and turning off filesystem `stat()` checks via `opcache.validate_timestamps=0`. • ✓ Build, verify, and ship Module Capstone: Cgroup-Aware Java 21 Layered JRE & Immutable OPcache PHP 8.3 Build Suite.

6.1 Containerizing Java 21 & Spring Boot — `MaxRAMPercentage`, Non-Heap Headroom & Layered JREs

[50 mins • Assignment (100M) • Hands-On Lab (50M) • 10 Mock Sets • 10 AI Vivas]

Subtopics Covered: JDK builder vs JRE / jlink runtime & -XX:+UseContainerSupport cgroup v2 • Core Syntax, CLI & Production APIs: eclipse-temurin:21-jre-jammy, -XX:+UseContainerSupport, -XX:MaxRAMPercentage=75.0, jlink • Deep-Dive Mechanics: -XX:MaxRAMPercentage=75.0 headroom & Heap + Metaspace + Direct NIO = RSS • Advanced Patterns: Spring Boot layered JAR cache split & ActiveProcessorCount CPU quota trap • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab: Multi-Stage Java 21 `eclipse-temurin` JRE Container with `MaxRAMPercentage` • Idiomatic DOCK Implementation: multi-stage BuildKit Dockerfiles, distroless/Alpine images & Compose networks • Linux Namespaces, cgroups v2, OverlayFS Layers & containerd/runc OCI Execution & Memory Lifecycle for Containerizing Java 21

Purpose: To understand and apply Containerizing Java 21 & Spring Boot — `MaxRAMPercentage`, Non-Heap Headroom & Layered JREs in real Docker systems — specifically: Why shipping a...

What Topic Is Next → Next Topic 6.2: Containerizing PHP 8.3 & Laravel — Multi-Stage Composer, `php-fpm` Tuning & Immutable `OPcache`

6.2 Containerizing PHP 8.3 & Laravel — Multi-Stage Composer, `php-fpm` Tuning & Immutable `OPcache`

[50 mins • Assignment (100M) • Hands-On Lab (50M) • 10 Mock Sets • 10 AI Vivas]

Subtopics Covered: COPY --from=composer:2 clean stage & --no-dev --classmap-authoritative • Core Syntax, CLI & Production APIs: FROM php:8.3-fpm-alpine, COPY --from=composer:2, docker-php-ext-install opcache pdo_mysql, p... • Deep-Dive Mechanics: Opcache.validate_timestamps=0 boost & Php-fpm + Nginx vs FrankenPHP • Advanced Patterns: Pm.max_children memory math & Www-data / non-root storage perms • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab: Multi-Stage PHP 8.3 CLI/FPM `Dockerfile` with Authoritative Composer & OPcache • Idiomatic DOCK Implementation: multi-stage BuildKit Dockerfiles, distroless/Alpine images & Compose networks • Linux Namespaces, cgroups v2, OverlayFS Layers & containerd/runc OCI Execution & Memory Lifecycle for Containerizing PHP 8.3

Purpose: To understand and apply Containerizing PHP 8.3 & Laravel — Multi-Stage Composer, `php-fpm` Tuning & Immutable `OPcache` in real Docker systems — specifically: Installing...

What Topic Is Next → Module 6 Capstone (Cgroup-Aware Java 21 Layered JRE & Immutable OPcache PHP 8.3 Build Suite) → Next Topic 7.1: Ubuntu & Debian Base Images — `DEBIAN_FRONTEND=noninteractive`, Single-Layer `apt-get` & Distroless Trade-Offs

Module 6 Hands-On Capstone Project (100 Marks): Cgroup-Aware Java 21 Layered JRE & Immutable OPcache PHP 8.3 Build Suite — Create production multi-stage `Dockerfile`s for (1) a Java 21 service protected against cgroup `OOMKilled 137` via `MaxRAMPercentage=75.0` and (2) an immutable PHP 8.3 FPM container with pre...

MODULE 7 OF 16 • STAGE 2 • 11 LIVE + LAB HOURS • 2 TOPICS (16 SUBTOPICS) • 1 CAPSTONE PROJECT (100 MARKS)

Module 7: Base Image Selection (`Ubuntu` / `Alpine` / `Distroless`) & `Overlay2` Storage Internals

Choose accurately between `ubuntu:24.04` (`glibc`), `alpine` (`musl`), and `distroless` base images, and understand Linux `overlay2` Copy-on-Write (`lowerdir`, `upperdir`, `merged`) vs persistent Docker Volumes.

Objectives: ✓ Ubuntu & Debian Base Images — `DEBIAN_FRONTEND=noninteractive`, Single-Layer `apt-get` & Distroless Trade-Offs — When full `glibc` compatibility (`ubuntu:24.04` ~29MB compressed / `debian:bookworm-slim`) beats `alpine` (`musl` DNS/threading quirks) or shell-less `distroless`, avoiding the cached `RUN apt-get update` anti-pattern, and non-interactive `tzdata` installs. • ✓ Docker Storage Drivers — `overlay2` (`lowerdir` / `upperdir` / `merged`), Copy-on-Write & Named Volumes — How Linux kernel `overlay2` unifies multiple read-only image layers (`lowerdir`) with the container's writable `upperdir` into a single `merged` view, the `copy_up` first-write penalty on large files, inode limits (`backingFsBlockDev` XFS `d_type=true` `ftype=1`), and why Docker Named Volumes bypass `overlay2` entirely. • ✓ Build, verify, and ship Module Capstone: CVE-Minimal Ubuntu/Distroless Base Benchmark & `Overlay2` vs Volume I/O Lab.

7.1 Ubuntu & Debian Base Images — `DEBIAN_FRONTEND=noninteractive`, Single-Layer `apt-get` & Distroless Trade-Offs

[50 mins • Assignment (100M) • Hands-On Lab (50M) • 10 Mock Sets • 10 AI Vivas]

Subtopics Covered: Atomic RUN apt-get update && install & ARG DEBIAN_FRONTEND=noninteractive • Core Syntax, CLI & Production APIs: FROM ubuntu:24.04, ARG DEBIAN_FRONTEND=noninteractive, apt-get install -y --no-install-recom... • Deep-Dive Mechanics: --no-install-recommends bloat cut & Rm -rf /var/lib/apt/lists/* in same RUN • Advanced Patterns: Glibc (Ubuntu/Debian) vs musl (Alpine) & Ubuntu 24.04 UID 1000 ubuntu user • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab: Hardened Non-Interactive `ubuntu:24.04` LTS Production `Dockerfile` • Idiomatic DOCK Implementation: multi-stage BuildKit Dockerfiles, distroless/Alpine images & Compose networks • Linux Namespaces, cgroups v2, OverlayFS Layers & containerd/runc OCI Execution & Memory Lifecycle for Ubuntu

Purpose: To understand and apply Ubuntu & Debian Base Images — `DEBIAN_FRONTEND=noninteractive`, Single-Layer `apt-get` & Distroless Trade-Offs in real Docker systems — specifical...

What Topic Is Next → Next Topic 7.2: Docker Storage Drivers — `overlay2` (`lowerdir` / `upperdir` / `merged`), Copy-on-Write & Named Volumes

7.2 Docker Storage Drivers — `overlay2` (`lowerdir` / `upperdir` / `merged`), Copy-on-Write & Named Volumes

[50 mins • Assignment (100M) • Hands-On Lab (50M) • 10 Mock Sets • 10 AI Vivas]

Subtopics Covered: Overlay2 lowerdir + upperdir = merged & First-write full-file copy_up cost • Core Syntax, CLI & Production APIs: overlay2 (lowerdir, upperdir, workdir, merged), docker volume create, --mount type=volume /... • Deep-Dive Mechanics: XFS ftype=1 (d_type) prerequisite & Named Volumes bypass overlay2 CoW • Advanced Patterns: Tmpfs RAM mounts for ephemeral secrets & Containerd snapshotter OCI store • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab: Inspecting `overlay2` vs Direct Volume Mount Topology in `compose.yaml` • Idiomatic DOCK Implementation: multi-stage BuildKit Dockerfiles, distroless/Alpine images & Compose networks • Linux Namespaces, cgroups v2, OverlayFS Layers & containerd/runc OCI Execution & Memory Lifecycle for Docker Storage Drivers

Purpose: To understand and apply Docker Storage Drivers — `overlay2` (`lowerdir` / `upperdir` / `merged`), Copy-on-Write & Named Volumes in real Docker systems — specifically: How...

What Topic Is Next → Module 7 Capstone (CVE-Minimal Ubuntu/Distroless Base Benchmark & `Overlay2` vs Volume I/O Lab) → Next Topic 8.1: Docker Compose v2 — `service_healthy` Readiness, Embedded `127.0.0.11` DNS, Profiles & Isolated Networks

Module 7 Hands-On Capstone Project (100 Marks): CVE-Minimal Ubuntu/Distroless Base Benchmark & `Overlay2` vs Volume I/O Lab — Build a non-interactive hardened Ubuntu 24.04 LTS utility container with zero leftover apt caches, and verify why PostgreSQL/MySQL data directories MUST reside on named Docker Volumes rather...

MODULE 8 OF 16 • STAGE 2 • 11 LIVE + LAB HOURS • 2 TOPICS (16 SUBTOPICS) • 1 CAPSTONE PROJECT (100 MARKS)

Module 8: Multi-Container `Docker Compose v2` Stacks & OCI Registry Supply-Chain Security

Wire multi-tier application stacks with `compose.yaml` (`condition: service_healthy`, isolated custom bridge DNS, Compose `secrets`), and publish multi-arch signed images to Docker Hub / GHCR / ECR.

Objectives: ✓ Docker Compose v2 — `service_healthy` Readiness, Embedded `127.0.0.11` DNS, Profiles & Isolated Networks — Modern `compose.yaml` (no obsolete `version:` header needed in Compose Spec!): automatic user-defined bridge network with embedded `127.0.0.11` service-name DNS, why plain `depends_on: [db]` only waits for container *process start* (NOT port readiness!), `condition: service_healthy`, `profiles`, and `docker compose watch`. • ✓ Pushing to OCI Registries — `--password-stdin`, Immutable SemVer+SHA Tags, Multi-Arch `buildx` & SBOMs — Registry naming (`<registry>/<namespace>/<repo>:<tag>`), piping PAT/OIDC tokens via `docker login --password-stdin`, why overwriting `:latest` breaks rollbacks, building Apple Silicon + x86_64 multi-arch manifests (`docker buildx build --platform linux/amd64,linux/arm64 --push`), and SBOM/provenance attestations. • ✓ Build, verify, and ship Module Capstone: 3-Tier Health-Gated `compose.yaml` Stack (`internal: true` DB Network) & Multi-Arch Buildx Push.

8.1 Docker Compose v2 — `service_healthy` Readiness, Embedded `127.0.0.11` DNS, Profiles & Isolated Networks

[50 mins • Assignment (100M) • Hands-On Lab (50M) • 10 Mock Sets • 10 AI Vivas]

Subtopics Covered: No legacy version: key in Compose v2 & Depends_on condition: service_healthy • Core Syntax, CLI & Production APIs: compose.yaml, docker compose up -d --wait, depends_on.condition: service_healthy, networks.<... • Deep-Dive Mechanics: Embedded 127.0.0.11 DNS on custom bridge & Internal: true air-gapped DB network • Advanced Patterns: Docker compose up -d --wait & Compose profiles & compose watch • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab: Healthcheck-Gated 2-Tier `compose.yaml` with Air-Gapped DB Network • Idiomatic DOCK Implementation: multi-stage BuildKit Dockerfiles, distroless/Alpine images & Compose networks • Linux Namespaces, cgroups v2, OverlayFS Layers & containerd/runc OCI Execution & Memory Lifecycle for Docker Compose v2

Purpose: To understand and apply Docker Compose v2 — `service_healthy` Readiness, Embedded `127.0.0.11` DNS, Profiles & Isolated Networks in real Docker systems — specifically: Mo...

What Topic Is Next → Next Topic 8.2: Pushing to OCI Registries — `--password-stdin`, Immutable SemVer+SHA Tags, Multi-Arch `buildx` & SBOMs

8.2 Pushing to OCI Registries — `--password-stdin`, Immutable SemVer+SHA Tags, Multi-Arch `buildx` & SBOMs

[50 mins • Assignment (100M) • Hands-On Lab (50M) • 10 Mock Sets • 10 AI Vivas]

Subtopics Covered: Registry/namespace/repo:tag taxonomy & Docker login --password-stdin safety • Core Syntax, CLI & Production APIs: docker login --username <u> --password-stdin, docker tag, docker push, docker buildx build -... • Deep-Dive Mechanics: Exec format error (arm64 vs amd64) & Docker buildx multi-arch manifest list • Advanced Patterns: SemVer + Git SHA immutable tagging & SBOM & Cosign provenance signing • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab: Multi-Architecture OCI Build & Digest Verification Workflow • Idiomatic DOCK Implementation: multi-stage BuildKit Dockerfiles, distroless/Alpine images & Compose networks • Linux Namespaces, cgroups v2, OverlayFS Layers & containerd/runc OCI Execution & Memory Lifecycle for Pushing to OCI Registries

Purpose: To understand and apply Pushing to OCI Registries — `--password-stdin`, Immutable SemVer+SHA Tags, Multi-Arch `buildx` & SBOMs in real Docker systems — specifically: Regi...

What Topic Is Next → Module 8 Capstone (3-Tier Health-Gated `compose.yaml` Stack (`internal: true` DB Network) & Multi-Arch Buildx Push) → Next Topic 9.1: Managed Cloud Containers — AWS ECS Fargate, Cloud Run, Pull-Through Cache Mirrors & Remote BuildKit

Module 8 Hands-On Capstone Project (100 Marks): 3-Tier Health-Gated `compose.yaml` Stack (`internal: true` DB Network) & Multi-Arch Buildx Push — Orchestrate a Web + API + PostgreSQL stack in `compose.yaml` where the API waits for `pg_isready` (`service_healthy`) and the database sits on an `internal: true` network with zero outbound...

MODULE 9 OF 16 • STAGE 3 • 10 LIVE + LAB HOURS • 2 TOPICS (16 SUBTOPICS) • 1 CAPSTONE PROJECT (100 MARKS)

Module 9: Managed Cloud Container Platforms & Docker vs Kubernetes Architecture Trade-Offs

Deploy containers to serverless/managed cloud runtimes (AWS ECS Fargate, Cloud Run, Docker Build Cloud, Registry Mirrors), and clarify how Docker (`containerd` packaging) complements Kubernetes orchestration.

Objectives: ✓ Managed Cloud Containers — AWS ECS Fargate, Cloud Run, Pull-Through Cache Mirrors & Remote BuildKit — Running OCI containers in production without managing EC2 hosts (AWS ECS Fargate, Google Cloud Run scale-to-zero, Azure Container Apps), avoiding Docker Hub anonymous rate limits (`100 pulls / 6 hrs / IP`) in cloud NAT Gateways via ECR/Artifact Registry pull-through caches, and remote `Docker Build Cloud` shared caches. • ✓ Docker vs Kubernetes — Packaging (`OCI` Build) vs Cluster Orchestration & Why `dockershim` Removal Changed Nothing for Dockerfiles — Clearing up the industry's #1 interview myth: **Docker** builds & runs OCI containers on a single machine (and local multi-container Compose stacks), whereas **Kubernetes** schedules, heals, autoscales, and networks thousands of those exact OCI containers across multi-node clusters via `containerd` CRI!. • ✓ Build, verify, and ship Module Capstone: Cloud-Ready 12-Factor Container Contract & Orchestrator Decision Matrix.

9.1 Managed Cloud Containers — AWS ECS Fargate, Cloud Run, Pull-Through Cache Mirrors & Remote BuildKit

[50 mins • Assignment (100M) • Hands-On Lab (50M) • 10 Mock Sets • 10 AI Vivas]

Subtopics Covered: AWS ECS Fargate & Cloud Run serverless & Docker Hub 429 NAT rate-limit trap • Core Syntax, CLI & Production APIs: AWS ECS TaskDefinition (Fargate), Google Cloud Run, ECR Pull-Through Cache, docker buildx --... • Deep-Dive Mechanics: ECR / GAR pull-through registry cache & Listen 0.0.0.0 inside

container netns • Advanced Patterns: 12-Factor stdout JSON log contract & Remote BuildKit shared team cache • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab: 12-Factor Cloud Run / Fargate Ready Container Spec (`0.0.0.0:$PORT`) • Idiomatic DOCK Implementation: multi-stage BuildKit Dockerfiles, distroless/Alpine images & Compose networks • Linux Namespaces, cgroups v2, OverlayFS Layers & containerd/runc OCI Execution & Memory Lifecycle for Managed Cloud Containers

Purpose: To understand and apply Managed Cloud Containers — AWS ECS Fargate, Cloud Run, Pull-Through Cache Mirrors & Remote BuildKit in real Docker systems — specifically: Running...

What Topic Is Next → Next Topic 9.2: Docker vs Kubernetes — Packaging (`OCI` Build) vs Cluster Orchestration & Why `dockershim` Removal Changed Nothing for Dockerfiles

9.2 Docker vs Kubernetes — Packaging (`OCI` Build) vs Cluster Orchestration & Why `dockershim` Removal Changed Nothing for Dockerfiles [50 mins • Assignment (100M) • Hands-On Lab (50M) • 10 Mock Sets • 10 AI Vivas]

Subtopics Covered: Docker (build/single-host) + K8s (cluster) & OCI standard bridges build to runtime • Core Syntax, CLI & Production APIs: docker build (OCI Image) -> OCI Registry -> Kubernetes kubelet -> CRI -> containerd -> runc • Deep-Dive Mechanics: Dockershim removal myth debunked & Kubelet -> CRI -> containerd direct path • Advanced Patterns: Compose vs Swarm vs Cloud Run vs K8s & Identical image SHA across dev & K8s • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab: Validating OCI Media-Type Compatibility for Kubernetes `containerd` • Idiomatic DOCK Implementation: multi-stage BuildKit Dockerfiles, distroless/Alpine images & Compose networks • Linux Namespaces, cgroups v2, OverlayFS Layers & containerd/runc OCI Execution & Memory Lifecycle for Docker vs Kubernetes

Purpose: To understand and apply Docker vs Kubernetes — Packaging (`OCI` Build) vs Cluster Orchestration & Why `dockershim` Removal Changed Nothing for Dockerfiles in real Docker...

What Topic Is Next → Module 9 Capstone (Cloud-Ready 12-Factor Container Contract & Orchestrator Decision Matrix) → Next Topic 10.1: Containerizing Ruby & Rails — Multi-Stage Bundler (`BUNDLE_DEPLOYMENT=1`), `libjemalloc2` & Puma

Module 9 Hands-On Capstone Project (100 Marks): Cloud-Ready 12-Factor Container Contract & Orchestrator Decision Matrix — Package a stateless 12-Factor container meeting strict Cloud Run / AWS Fargate / Kubernetes runtime contracts (`PORT` env binding, stdout structured JSON logging, `<10s` graceful `SIGTERM` d...

MODULE 10 OF 16 • STAGE 3 • 10 LIVE + LAB HOURS • 2 TOPICS (16 SUBTOPICS) • 1 CAPSTONE PROJECT (100 MARKS)

Module 10: Polyglot Production Packaging — `Ruby on Rails` (`Jemalloc`/Puma) & `Scala` (`sbt` Layered JVM)

Complete your polyglot container mastery by packaging memory-efficient Ruby 3.3 / Rails 7+ (`jemalloc` + Puma) and fast-caching Scala 3 / `sbt` multi-stage JVM microservices.

Objectives: ✓ Containerizing Ruby & Rails — Multi-Stage Bundler (`BUNDLE_DEPLOYMENT=1`), `libjemalloc2` & Puma — Why compiling native C gems (`pg`, `nokogiri`, `ffi`) requires `build-essential` + `libpq-dev` ONLY in the builder stage while the slim runtime needs only `libpq5` + `libjemalloc2`, freezing `Gemfile.lock` with `BUNDLE_DEPLOYMENT=1`, and `RAILS_LOG_TO_STDOUT=1`. • ✓ Containerizing Scala & `sbt` — Coursier Cache Mounts, `sbt stage` vs Fat `assembly` JARs & JRE Runtime — Why shipping an `sbt` + JDK builder image produces a bloated `1.4 GB` container, caching `project/build.properties` + `project/plugins.sbt` + `build.sbt` with BuildKit Coursier cache mounts, and preferring `sbt-native-packager` (`sbt stage`) layered classpaths over slow monolithic `sbt assembly` uber-JAR merge conflicts. • ✓ Build, verify, and ship Module Capstone: Polyglot Multi-Stage Build Suite (`Ruby 3.3 + jemalloc` & `Scala 3 sbt-native-packager`).

10.1 Containerizing Ruby & Rails — Multi-Stage Bundler (`BUNDLE_DEPLOYMENT=1`), `libjemalloc2` & Puma [50 mins • Assignment (100M) • Hands-On Lab (50M) • 10 Mock Sets • 10 AI Vivas]

Subtopics Covered: Gemfile.lock-first layer caching & BUNDLE_DEPLOYMENT=1 lockfile guard • Core Syntax, CLI & Production APIs: FROM ruby:3.3-slim, BUNDLE_DEPLOYMENT="1", BUNDLE_WITHOUT="development:test", LD_PRELOAD=/us... • Deep-Dive Mechanics: Build-essential in builder stage only & Libjemalloc2 RSS fragmentation cut • Advanced Patterns: RAILS_LOG_TO_STDOUT=1 12-Factor logs & Puma exec-form SIGTERM phased shutdown • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab: Multi-Stage Ruby 3.3 + `jemalloc` Non-Root Production `Dockerfile` • Idiomatic DOCK Implementation: multi-stage BuildKit Dockerfiles, distroless/Alpine images & Compose networks • Linux Namespaces, cgroups v2, OverlayFS Layers & containerd/runc OCI Execution & Memory Lifecycle for Containerizing Ruby

Purpose: To understand and apply Containerizing Ruby & Rails — Multi-Stage Bundler (`BUNDLE_DEPLOYMENT=1`), `libjemalloc2` & Puma in real Docker systems — specifically: Why compil...

What Topic Is Next → Next Topic 10.2: Containerizing Scala & `sbt` — Coursier Cache Mounts, `sbt stage` vs Fat `assembly` JARs & JRE Runtime

10.2 Containerizing Scala & `sbt` — Coursier Cache Mounts, `sbt stage` vs Fat `assembly` JARs & JRE Runtime [50 mins • Assignment (100M) • Hands-On Lab (50M) • 10 Mock Sets • 10 AI Vivas]

Subtopics Covered: Build.sbt + project/ cache layer first & Coursier / Ivy BuildKit cache mount • Core Syntax, CLI & Production APIs: sbt-native-packager (sbt stage), --mount=type=cache,target=/root/.cache/coursier, eclipse-te... • Deep-Dive Mechanics: Sbt-native-packager stage vs assembly & Reference.conf merge-conflict safety • Advanced Patterns: Strip 1.3GB sbt+JDK via JRE stage 2 & -XX:MaxRAMPercentage=75.0 JVM cgroups • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab: Multi-Stage Scala 3 (`sbt stage` -> Non-Root Java 21 JRE) `Dockerfile` • Idiomatic DOCK Implementation: multi-stage BuildKit Dockerfiles, distroless/Alpine images & Compose networks • Linux Namespaces, cgroups v2, OverlayFS Layers & containerd/runc OCI Execution & Memory Lifecycle for Containerizing Scala

Purpose: To understand and apply Containerizing Scala & `sbt` — Coursier Cache Mounts, `sbt stage` vs Fat `assembly` JARs & JRE Runtime in real Docker systems — specifically: Why...

What Topic Is Next → Module 10 Capstone (Polyglot Multi-Stage Build Suite (`Ruby 3.3 + jemalloc` & `Scala 3 sbt-native-packager`)) → Next Topic 11.1: BuildKit Cache Mounts (--mount=type=cache) & Build Secret Mounts

Module 10 Hands-On Capstone Project (100 Marks): Polyglot Multi-Stage Build Suite (`Ruby 3.3 + jemalloc` & `Scala 3 sbt-native-packager`) — Author production multi-stage `Dockerfile`s for a Ruby 3.3 Puma web service using `jemalloc` to slash heap fragmentation by ~35% and a Scala 3 service stripping `sbt` from the runtime image.

MODULE 11 OF 16 • STAGE 3 • 16 LIVE + LAB HOURS • 4 TOPICS (32 SUBTOPICS) • 1 CAPSTONE PROJECT (100 MARKS)

BuildKit Advanced Caching, Multi-Arch (AMD64/ARM64) & Distroless Images

Production-grade Docker Training engineering module covering BuildKit Advanced Caching, Multi-Arch (AMD64/ARM64) & Distroless Images with hands-on labs and capstone build.

Objectives: ✓ BuildKit Cache Mounts (--mount=type=cache) & Build Secret Mounts — Purpose: Speed up npm/pip/Maven builds 6x without ever baking private tokens into image layers!. • ✓ Multi-Stage Builds into Distroless / Scratch / Alpine Runtimes — Purpose: Strip compilers and shells out of prod images, slashing size and CVE surface by 85%. • ✓ Cross-Compiling linux/amd64 + linux/arm64 Images with docker buildx — Purpose: Run identical container tags on Apple Silicon laptops and AWS Graviton servers. • ✓ Layer Ordering Discipline & Strict .dockerignore Hygiene — Purpose: Ensure editing one source file never invalidates cached dependency layers.

11.1 BuildKit Cache Mounts (--mount=type=cache) & Build Secret Mounts [65 mins • Assignment (100M) • Hands-On Lab (50M) • 2 Mock Sets • 2 AI Vivas]

Subtopics Covered: Plain • Production Implementation of Build Secret Mounts • Integration with Multi • Live Debugging, Telemetry & Technical Interview Trade • What Is BuildKit Cache Mounts (• In Docker Training, BuildKit Cache Mounts (• Even if you come from a non • Runnable Code Implementation (DOCKERFILE)

Purpose: To understand and apply BuildKit Cache Mounts (--mount=type=cache) & Build Secret Mounts in real Docker systems — specifically: Speed up npm/pip/Maven builds 6x without e...

What Topic Is Next → Next Topic 11.2: Multi-Stage Builds into Distroless / Scratch / Alpine Runtimes

11.2 Multi-Stage Builds into Distroless / Scratch / Alpine Runtimes [65 mins • Assignment (100M) • Hands-On Lab (50M) • 2 Mock Sets • 2 AI Vivas]

Subtopics Covered: Plain • Production Implementation of Edge • Integration with Cross • Live Debugging, Telemetry & Technical Interview Trade • What Is Multi • In Docker Training, Multi • Even if you come from a non • Runnable Code Implementation (DOCKERFILE)

Purpose: To understand and apply Multi-Stage Builds into Distroless / Scratch / Alpine Runtimes in real Docker systems — specifically: Strip compilers and shells out of prod image...

What Topic Is Next → Next Topic 11.3: Cross-Compiling linux/amd64 + linux/arm64 Images with docker buildx

11.3 Cross-Compiling linux/amd64 + linux/arm64 Images with docker buildx [65 mins • Assignment (100M) • Hands-On Lab (50M) • 2 Mock Sets • 2 AI Vivas]

Subtopics Covered: Plain • Production Implementation of Edge • Integration with Layer Ordering Discipline & Strict .dockerignore Hygiene • Live Debugging, Telemetry & Technical Interview Trade • What Is Cross • In Docker Training, Cross • Even if you come from a non • Runnable Code Implementation (DOCKERFILE)

Purpose: To understand and apply Cross-Compiling linux/amd64 + linux/arm64 Images with docker buildx in real Docker systems — specifically: Run identical container tags on Apple S...

What Topic Is Next → Next Topic 11.4: Layer Ordering Discipline & Strict .dockerignore Hygiene

11.4 Layer Ordering Discipline & Strict .dockerignore Hygiene [65 mins • Assignment (100M) • Hands-On Lab (50M) • 2 Mock Sets • 2 AI Vivas]

Subtopics Covered: Plain • Production Implementation of Strict .dockerignore Hygiene • Integration with BuildKit Cache Mounts (• Live Debugging, Telemetry & Technical Interview Trade • What Is Layer Ordering Discipline & Strict .dockerignore Hygiene & Why Industry Teams Use It • In Docker Training, Layer Ordering Discipline & Strict .dockerignore Hygiene exists for one clear real • Even if you come from a non • Runnable Code Implementation (DOCKERFILE)

Purpose: To understand and apply Layer Ordering Discipline & Strict .dockerignore Hygiene in real Docker systems — specifically: Ensure editing one source file never invalidates c...

What Topic Is Next → Module 11 Capstone (Sub-50MB Multi-Arch Production Container Build Pipeline) → Next Topic 12.1: Linux Namespaces, Control Groups (cgroups v2) & OverlayFS Internals

Module 11 Hands-On Capstone Project (100 Marks): Sub-50MB Multi-Arch Production Container Build Pipeline — Design, implement, test, and document 'Sub-50MB Multi-Arch Production Container Build Pipeline' applying every production concept from Module 11 (BuildKit Advanced Caching, Multi-Arch (AMD64...

MODULE 12 OF 16 • STAGE 3 • 16 LIVE + LAB HOURS • 4 TOPICS (32 SUBTOPICS) • 1 CAPSTONE PROJECT (100 MARKS)

Container Security Hardening: Linux Namespaces, cgroups, Trivy & SBOMs

Production-grade Docker Training engineering module covering Container Security Hardening: Linux Namespaces, cgroups, Trivy & SBOMs with hands-on labs and capstone build.

Objectives: ✓ Linux Namespaces, Control Groups (cgroups v2) & OverlayFS Internals — Purpose: Understand what containers actually are under the hood on the Linux kernel. • ✓ Non-Root USER, --cap-drop=ALL, --read-only & no-new-privileges — Purpose: Contain exploit blast radius even if an application library has a zero-day bug. • ✓ Automated Vulnerability Scanning with Trivy & Syft CycloneDX SBOMs — Purpose: Fail GitHub Actions builds automatically when Critical OS/library CVEs appear. • ✓ Cosign Keyless Image Signing & Private Registry Access Controls — Purpose: Verify cryptographic provenance so clusters run only trusted CI-built images.

12.1 Linux Namespaces, Control Groups (cgroups v2) & OverlayFS Internals [65 mins • Assignment (100M) • Hands-On Lab (50M) • 2 Mock Sets • 2 AI Vivas]

Subtopics Covered: Plain • Production Implementation of Control Groups (cgroups v2) • Integration with Non • Live Debugging, Telemetry & Technical Interview Trade • What Is Linux Namespaces, Control Groups (cgroups v2) & OverlayFS Internals & Why Industry Teams Use It • In Docker Training, Linux Namespaces, Control Groups (cgroups v2) & OverlayFS Internals exists for one clear real • Even if you come from a non • Runnable Code Implementation (DOCKERFILE)

Purpose: To understand and apply Linux Namespaces, Control Groups (cgroups v2) & OverlayFS Internals in real Docker systems — specifically: Understand what containers actually are...

What Topic Is Next → Next Topic 12.2: Non-Root USER, --cap-drop=ALL, --read-only & no-new-privileges

12.2 Non-Root USER, --cap-drop=ALL, --read-only & no-new-privileges [65 mins • Assignment (100M) • Hands-On Lab (50M) • 2 Mock Sets • 2 AI Vivas]

Subtopics Covered: Plain • Production Implementation of • Integration with Automated Vulnerability Scanning with Trivy & Syft CycloneDX SBOMs • Live Debugging, Telemetry & Technical Interview Trade • What Is Non • In Docker Training, Non • Even if you come from a non • Runnable Code Implementation (DOCKERFILE)

Purpose: To understand and apply Non-Root USER, --cap-drop=ALL, --read-only & no-new-privileges in real Docker systems — specifically: Contain exploit blast radius even if an appl...

What Topic Is Next → Next Topic 12.3: Automated Vulnerability Scanning with Trivy & Syft CycloneDX SBOMs

12.3 Automated Vulnerability Scanning with Trivy & Syft CycloneDX SBOMs [65 mins • Assignment (100M) • Hands-On Lab (50M) • 2 Mock Sets • 2 AI Vivas]

Subtopics Covered: Plain • Production Implementation of Syft CycloneDX SBOMs • Integration with Cosign Keyless Image Signing & Private Registry Access Controls • Live Debugging, Telemetry & Technical Interview Trade • What Is Automated Vulnerability Scanning with Trivy & Syft CycloneDX SBOMs & Why Industry Teams Use It • In Docker Training, Automated Vulnerability Scanning with Trivy & Syft CycloneDX SBOMs exists for one clear real • Even if you come from a non • Runnable Code Implementation (DOCKERFILE)

Purpose: To understand and apply Automated Vulnerability Scanning with Trivy & Syft CycloneDX SBOMs in real Docker systems — specifically: Fail GitHub Actions builds automatically...

What Topic Is Next → Next Topic 12.4: Cosign Keyless Image Signing & Private Registry Access Controls

12.4 Cosign Keyless Image Signing & Private Registry Access Controls [65 mins • Assignment (100M) • Hands-On Lab (50M) • 2 Mock Sets • 2 AI Vivas]

Subtopics Covered: Plain • Production Implementation of Private Registry Access Controls • Integration with Linux Namespaces, Control Groups (cgroups v2) & OverlayFS Internals • Live Debugging, Telemetry & Technical Interview Trade • What Is Cosign Keyless Image Signing & Private Registry Access Controls & Why Industry Teams Use It • In Docker Training, Cosign Keyless Image Signing & Private Registry Access Controls exists for one clear real • Even if you come from a non • Runnable Code Implementation (DOCKERFILE)

Purpose: To understand and apply Cosign Keyless Image Signing & Private Registry Access Controls in real Docker systems — specifically: Verify cryptographic provenance so clusters...

What Topic Is Next → Module 12 Capstone (Zero-Critical-CVE Hardened Non-Root Container Supply Chain) → Next Topic 13.1: depends_on: service_healthy with Real DB/Redis Readiness Checks

Module 12 Hands-On Capstone Project (100 Marks): Zero-Critical-CVE Hardened Non-Root Container Supply Chain — Design, implement, test, and document 'Zero-Critical-CVE Hardened Non-Root Container Supply Chain' applying every production concept from Module 12 (Container Security Hardening: Linux Names...

MODULE 13 OF 16 • STAGE 4 • 16 LIVE + LAB HOURS • 4 TOPICS (32 SUBTOPICS) • 1 CAPSTONE PROJECT (100 MARKS)

Production Docker Compose: Healthchecks, Isolated Networks, TLS & Limits

Production-grade Docker Training engineering module covering Production Docker Compose: Healthchecks, Isolated Networks, TLS & Limits with hands-on labs and capstone build.

Objectives: ✓ depends_on: service_healthy with Real DB/Redis Readiness Checks — Purpose: Stop API containers from crashing at boot before PostgreSQL finishes initializing. • ✓ Internal-Only Backend Bridge Networks vs Public Reverse-Proxy Network — Purpose: Keep PostgreSQL and Redis completely unreachable from public host ports. • ✓ Compose Secrets, Named Volume Backups & CPU/Memory Resource Limits — Purpose: Prevent one runaway container from starving host RAM or losing stateful data. • ✓ PID 1 Signal Forwarding (tini / exec Form) for Clean SIGTERM Shutdown — Purpose: Ensure containers stop in 1 second gracefully instead of suffering 10s SIGKILL kills.

13.1 depends_on: service_healthy with Real DB/Redis Readiness Checks [65 mins • Assignment (100M) • Hands-On Lab (50M) • 2 Mock Sets • 2 AI Vivas]

Subtopics Covered: Plain • Production Implementation of Edge • Integration with Internal • Live Debugging, Telemetry & Technical Interview Trade • What Is depends_on • In Docker Training, depends_on • Even if you come from a non • Runnable Code Implementation (DOCKERFILE)

Purpose: To understand and apply depends_on: service_healthy with Real DB/Redis Readiness Checks in real Docker systems — specifically: Stop API containers from crashing at boot b...

What Topic Is Next → Next Topic 13.2: Internal-Only Backend Bridge Networks vs Public Reverse-Proxy Network

13.2 Internal-Only Backend Bridge Networks vs Public Reverse-Proxy Network [65 mins • Assignment (100M) • Hands-On Lab (50M) • 2 Mock Sets • 2 AI Vivas]

Subtopics Covered: Plain • Production Implementation of Edge • Integration with Compose Secrets, Named Volume Backups & CPU/Memory Resource Limits • Live Debugging, Telemetry & Technical Interview Trade • What Is Internal • In Docker Training, Internal • Even if you come from a non • Runnable Code Implementation (DOCKERFILE)

Purpose: To understand and apply Internal-Only Backend Bridge Networks vs Public Reverse-Proxy Network in real Docker systems — specifically: Keep PostgreSQL and Redis completely...

What Topic Is Next → Next Topic 13.3: Compose Secrets, Named Volume Backups & CPU/Memory Resource Limits

13.3 Compose Secrets, Named Volume Backups & CPU/Memory Resource Limits [65 mins • Assignment (100M) • Hands-On Lab (50M) • 2 Mock Sets • 2 AI Vivas]

Subtopics Covered: Plain • Production Implementation of Named Volume Backups • Integration with PID 1 Signal Forwarding (tini / exec Form) for Clean SIGTERM Shutdown • Live Debugging, Telemetry & Technical Interview Trade • What Is Compose Secrets, Named Volume Backups & CPU/Memory Resource Limits & Why Industry Teams Use It • In Docker Training, Compose Secrets, Named Volume Backups & CPU/Memory Resource Limits exists for one clear real • Even if you come from a non • Runnable Code Implementation (DOCKERFILE)

Purpose: To understand and apply Compose Secrets, Named Volume Backups & CPU/Memory Resource Limits in real Docker systems — specifically: Prevent one runaway container from starv...

What Topic Is Next → Next Topic 13.4: PID 1 Signal Forwarding (tini / exec Form) for Clean SIGTERM Shutdown

13.4 PID 1 Signal Forwarding (tini / exec Form) for Clean SIGTERM Shutdown [65 mins • Assignment (100M) • Hands-On Lab (50M) • 2 Mock Sets • 2 AI Vivas]

Subtopics Covered: Plain • Production Implementation of Edge • Integration with depends_on • Live Debugging, Telemetry & Technical Interview Trade • What Is PID 1 Signal Forwarding (tini / exec Form) for Clean SIGTERM Shutdown & Why Industry Teams Use It • In Docker Training, PID 1 Signal Forwarding (tini / exec Form) for Clean SIGTERM Shutdown exists for one clear real • Even if you come from a non • Runnable Code Implementation (DOCKERFILE)

Purpose: To understand and apply PID 1 Signal Forwarding (tini / exec Form) for Clean SIGTERM Shutdown in real Docker systems — specifically: Ensure containers stop in 1 second gr...

What Topic Is Next → Module 13 Capstone (Multi-Service App + DB + Redis + Caddy/Nginx TLS Stack) → Next Topic 14.1: Log Rotation (json-file max-size), Fluentd/Loki Drivers & Prometheus cAdvisor

Module 13 Hands-On Capstone Project (100 Marks): Multi-Service App + DB + Redis + Caddy/Nginx TLS Stack — Design, implement, test, and document 'Multi-Service App + DB + Redis + Caddy/Nginx TLS Stack' applying every production concept from Module 13 (Production Docker Compose: Healthchecks, Isol...

MODULE 14 OF 16 • STAGE 4 • 16 LIVE + LAB HOURS • 4 TOPICS (32 SUBTOPICS) • 1 CAPSTONE PROJECT (100 MARKS)

Container Observability, Log Drivers, Live Forensics & CI/CD Rollouts

Production-grade Docker Training engineering module covering Container Observability, Log Drivers, Live Forensics & CI/CD Rollouts with hands-on labs and capstone build.

Objectives: ✓ Log Rotation (json-file max-size), Fluentd/Loki Drivers & Prometheus cAdvisor — Purpose: Prevent unrotated container stdout logs from filling 100% of server disks!. • ✓ Live Container Diagnostics (docker stats, inspect, events & nsenter) — Purpose: Debug high memory, DNS resolution, and socket exhaustion on live containers. • ✓ GitHub Actions Registry Caching, Semantic SHA Tags & Rollback Scripts — Purpose: Eliminate mutable :latest tag chaos with immutable commit-SHA deployments. • ✓ Reverse-Proxy Blue/Green Container Cutover & Automated Prune Policies — Purpose: Swap traffic to verified new containers seamlessly and reclaim disk safely.

14.1 Log Rotation (json-file max-size), Fluentd/Loki Drivers & Prometheus cAdvisor [65 mins • Assignment (100M) • Hands-On Lab (50M) • 2 Mock Sets • 2 AI Vivas]

Subtopics Covered: Plain • Production Implementation of Fluentd/Loki Drivers • Integration with Live Container Diagnostics (docker stats, inspect, events & nsenter) • Live Debugging, Telemetry & Technical Interview Trade • What Is Log Rotation (json • In Docker Training, Log Rotation (json • Even if you come from a non • Runnable Code Implementation (DOCKERFILE)

Purpose: To understand and apply Log Rotation (json-file max-size), Fluentd/Loki Drivers & Prometheus cAdvisor in real Docker systems — specifically: Prevent unrotated container s...

What Topic Is Next → Next Topic 14.2: Live Container Diagnostics (docker stats, inspect, events & nsenter)

14.2 Live Container Diagnostics (docker stats, inspect, events & nsenter) [65 mins • Assignment (100M) • Hands-On Lab (50M) • 2 Mock Sets • 2 AI Vivas]

Subtopics Covered: Plain • Production Implementation of inspect • Integration with GitHub Actions Registry Caching, Semantic SHA Tags & Rollback Scripts • Live Debugging, Telemetry & Technical Interview Trade • What Is Live Container Diagnostics (docker stats, inspect, events & nsenter) & Why Industry Teams Use It • In Docker Training, Live Container Diagnostics (docker stats, inspect, events & nsenter) exists for one clear real • Even if you come from a non • Runnable Code Implementation (DOCKERFILE)

Purpose: To understand and apply Live Container Diagnostics (docker stats, inspect, events & nsenter) in real Docker systems — specifically: Debug high memory, DNS resolution, and...

What Topic Is Next → Next Topic 14.3: GitHub Actions Registry Caching, Semantic SHA Tags & Rollback Scripts

14.3 GitHub Actions Registry Caching, Semantic SHA Tags & Rollback Scripts [65 mins • Assignment (100M) • Hands-On Lab (50M) • 2 Mock Sets • 2 AI Vivas]

Subtopics Covered: Plain • Production Implementation of Semantic SHA Tags • Integration with Reverse • Live Debugging, Telemetry & Technical Interview Trade • What Is GitHub Actions Registry Caching, Semantic SHA Tags & Rollback Scripts & Why Industry Teams Use It • In Docker Training, GitHub Actions Registry Caching, Semantic SHA Tags & Rollback Scripts exists for one clear real • Even if you come from a non • Runnable Code Implementation (DOCKERFILE)

Purpose: To understand and apply GitHub Actions Registry Caching, Semantic SHA Tags & Rollback Scripts in real Docker systems — specifically: Eliminate mutable :latest tag chaos w...

What Topic Is Next → Next Topic 14.4: Reverse-Proxy Blue/Green Container Cutover & Automated Prune Policies

14.4 Reverse-Proxy Blue/Green Container Cutover & Automated Prune Policies [65 mins • Assignment (100M) • Hands-On Lab (50M) • 2 Mock Sets • 2 AI Vivas]

Subtopics Covered: Plain • Production Implementation of Automated Prune Policies • Integration with Log Rotation (json • Live Debugging, Telemetry & Technical Interview Trade • What Is Reverse • In Docker Training, Reverse • Even if you come from a non • Runnable Code Implementation (DOCKERFILE)

Purpose: To understand and apply Reverse-Proxy Blue/Green Container Cutover & Automated Prune Policies in real Docker systems — specifically: Swap traffic to verified new containe...

What Topic Is Next → Module 14 Capstone (Monitored Zero-Downtime Container Host with Centralized Logs) → Next Topic 15.1: BuildKit Cache Mounts, Secret Mounts (--mount=type=secret) & Layer-Order Mastery

Module 14 Hands-On Capstone Project (100 Marks): Monitored Zero-Downtime Container Host with Centralized Logs — Design, implement, test, and document 'Monitored Zero-Downtime Container Host with Centralized Logs' applying every production concept from Module 14 (Container Observability, Log Drivers, L...

MODULE 15 OF 16 • STAGE 4 • 14 LIVE + LAB HOURS • 3 TOPICS (24 SUBTOPICS) • 1 CAPSTONE PROJECT (100 MARKS)

Supply-Chain Security: BuildKit Secrets, SBOMs, Trivy Scanning & Distroless

Self-explanatory list-and-bullet production mastery module covering Supply-Chain Security: BuildKit Secrets, SBOMs, Trivy Scanning & Distroless in Docker Training, paired with hands-on labs, capstone repository, Topic Mock Tests, and Spoken MockAttempt AI Viva.

Objectives: ✓ BuildKit Cache Mounts, Secret Mounts (--mount=type=secret) & Layer-Order Mastery — Purpose: Cut CI Docker build times by 70% while ensuring private tokens never bake into image layers. • ✓ Distroless / Wolfi / Alpine Minimal Images, Non-Root UIDs & Dropped Linux Capabilities — Purpose: Shrink attack surface and image megabytes by stripping shells and package managers from runtime. • ✓ Vulnerability Scanning (Trivy), SBOM Generation & Signed Multi-Arch (amd64/arm64) Releases — Purpose: Block vulnerable base images in CI and build images that run on both x86 and AWS Graviton. • ✓ Build, verify, and document 'Zero-Critical-CVE Multi-Arch Distroless Image Pipeline with BuildKit Cache' with clean Git commits and edge-case protection. • ✓ Clear 100% of Topic Timed Mock Tests and Spoken Voice + MockAttempt AI Technical Interviews.

15.1 BuildKit Cache Mounts, Secret Mounts (--mount=type=secret) & Layer-Order Mastery [65 mins • Assignment (100M) • Hands-On Lab (50M) • 2 Mock Sets • 2 AI Vivas]

Subtopics Covered: Security Hygiene & Data Safety Rules for BuildKit Cache Mounts, Secret Mounts (• Performance Optimization, Complexity & Latency Impact of BuildKit Cache Mounts, Secret Mounts (• Live Debugging, Stack • Runnable Code Implementation (DOCKERFILE) • Hands • Idiomatic DOCK Implementation: multi-stage BuildKit Dockerfiles, distroless/Alpine images & Compose networks • Linux Namespaces, cgroups v2, OverlayFS Layers & containerd/runc OCI Execution & Memory Lifecycle for BuildKit Cache Mounts • Production Hardening & Security: non-root USER, read-only rootfs, dropped capabilities & Trivy CVE gates

Purpose: To understand and apply BuildKit Cache Mounts, Secret Mounts (--mount=type=secret) & Layer-Order Mastery in real Docker systems — specifically: Cut CI Docker build times...

What Topic Is Next → Next Topic 15.2: Distroless / Wolfi / Alpine Minimal Images, Non-Root UIDs & Dropped Linux Capabilities

15.2 Distroless / Wolfi / Alpine Minimal Images, Non-Root UIDs & Dropped Linux Capabilities [65 mins • Assignment (100M) • Hands-On Lab (50M) • 2 Mock Sets • 2 AI Vivas]

Subtopics Covered: Security Hygiene & Data Safety Rules for Distroless / Wolfi / Alpine Minimal Images, Non • Performance Optimization, Complexity & Latency Impact of Distroless / Wolfi / Alpine Minimal Images, Non • Live Debugging, Stack • Runnable Code Implementation (DOCKERFILE) • Hands • Idiomatic DOCK Implementation: multi-stage BuildKit Dockerfiles, distroless/Alpine images & Compose networks • Linux Namespaces, cgroups v2, OverlayFS Layers & containerd/runc OCI Execution & Memory Lifecycle for Distroless / Wolfi / Alpine Minimal Images • Production Hardening & Security: non-root USER, read-only rootfs, dropped capabilities & Trivy CVE gates

Purpose: To understand and apply Distroless / Wolfi / Alpine Minimal Images, Non-Root UIDs & Dropped Linux Capabilities in real Docker systems — specifically: Shrink attack surfac...

What Topic Is Next → Next Topic 15.3: Vulnerability Scanning (Trivy), SBOM Generation & Signed Multi-Arch (amd64/arm64) Releases

15.3 Vulnerability Scanning (Trivy), SBOM Generation & Signed Multi-Arch (amd64/arm64) Releases [65 mins • Assignment (100M) • Hands-On Lab (50M) • 2 Mock Sets • 2 AI Vivas]

Subtopics Covered: Security Hygiene & Data Safety Rules for Vulnerability Scanning (Trivy), SBOM Generation & Signed Multi • Performance Optimization, Complexity & Latency Impact of Vulnerability Scanning (Trivy), SBOM Generation & Signed M... • Live Debugging, Stack • Runnable Code Implementation (DOCKERFILE) • Hands • Idiomatic DOCK Implementation: multi-stage BuildKit Dockerfiles, distroless/Alpine images & Compose networks • Linux Namespaces, cgroups v2, OverlayFS Layers & containerd/runc OCI Execution & Memory Lifecycle for Vulnerability Scanning (Trivy) • Production Hardening & Security: non-root USER, read-only rootfs, dropped capabilities & Trivy CVE gates

Purpose: To understand and apply Vulnerability Scanning (Trivy), SBOM Generation & Signed Multi-Arch (amd64/arm64) Releases in real Docker systems — specifically: Block vulnerable...

What Topic Is Next → Module 15 Capstone (Zero-Critical-CVE Multi-Arch Distroless Image Pipeline with BuildKit Cache) → Next Topic 16.1: Healthchecks, service_completed_successfully Dependencies & Graceful PID 1 Signals

Module 15 Hands-On Capstone Project (100 Marks): Zero-Critical-CVE Multi-Arch Distroless Image Pipeline with BuildKit Cache — Architect, implement, test, and document 'Zero-Critical-CVE Multi-Arch Distroless Image Pipeline with BuildKit Cache' applying every bullet-wise engineering concept from Module 15 (Supply-Ch...

MODULE 16 OF 16 • STAGE 4 • 14 LIVE + LAB HOURS • 3 TOPICS (24 SUBTOPICS) • 1 CAPSTONE PROJECT (100 MARKS)

Production Docker Compose Orchestration, Resource Limits & Container Triage

Self-explanatory list-and-bullet production mastery module covering Production Docker Compose Orchestration, Resource Limits & Container Triage in Docker Training, paired with hands-on labs, capstone repository, Topic Mock Tests, and Spoken MockAttempt AI Viva.

Objectives: ✓ Healthchecks, service_completed_successfully Dependencies & Graceful PID 1 Signals — Purpose: Ensure apps wait for ready databases and receive SIGTERM cleanly without 10-second SIGKILL hangs. • ✓ Memory/CPU Cgroups Limits, Log Rotation Drivers & Isolated Internal Networks — Purpose: Prevent a single runaway container from exhausting host disk space or host RAM. • ✓ Container Filesystem/Overlay2 Internals & Docker Troubleshooting Interviews — Purpose: Inspect namespaces, cgroups, zombie processes, and DNS resolution issues confidently. • ✓ Build, verify, and document 'Health-Gated Reverse-Proxy + App + DB Compose Stack with Log/Memory Limits' with clean Git commits and edge-case protection. • ✓ Clear 100% of Topic Timed Mock Tests and Spoken Voice + MockAttempt AI Technical Interviews.

16.1 Healthchecks, service_completed_successfully Dependencies & Graceful PID 1 Signals [65 mins • Assignment (100M) • Hands-On Lab (50M) • 2 Mock Sets • 2 AI Vivas]

Subtopics Covered: Security Hygiene & Data Safety Rules for Healthchecks, service_completed_successfully Dependencies & Graceful PID 1... • Performance Optimization, Complexity & Latency Impact of Healthchecks, service_completed_successfully Dependencies.... • Live Debugging, Stack • Runnable Code Implementation (DOCKERFILE) • Hands • Idiomatic DOCK Implementation: multi-stage BuildKit Dockerfiles, distroless/Alpine images & Compose networks • Linux Namespaces, cgroups v2, OverlayFS Layers & containerd/runc OCI Execution & Memory Lifecycle for Healthchecks • Production Hardening & Security: non-root USER,

read-only rootfs, dropped capabilities & Trivy CVE gates

Purpose: To understand and apply Healthchecks, service_completed_successfully Dependencies & Graceful PID 1 Signals in real Docker systems — specifically: Ensure apps wait for rea...

What Topic Is Next → Next Topic 16.2: Memory/CPU Cgroups Limits, Log Rotation Drivers & Isolated Internal Networks

16.2 Memory/CPU Cgroups Limits, Log Rotation Drivers & Isolated Internal Networks [65 mins • Assignment (100M) • Hands-On Lab (50M) • 2 Mock Sets • 2 AI Vivas]

Subtopics Covered: Security Hygiene & Data Safety Rules for Memory/CPU Cgroups Limits, Log Rotation Drivers & Isolated Internal Networ... • Performance Optimization, Complexity & Latency Impact of Memory/CPU Cgroups Limits, Log Rotation Drivers & Isolated... • Live Debugging, Stack • Runnable Code Implementation (DOCKERFILE) • Hands • Idiomatic DOCK Implementation: multi-stage BuildKit Dockerfiles, distroless/Alpine images & Compose networks • Linux Namespaces, cgroups v2, OverlayFS Layers & containerd/runc OCI Execution & Memory Lifecycle for Memory/CPU Cgroups Limits • Production Hardening & Security: non-root USER, read-only rootfs, dropped capabilities & Trivy CVE gates

Purpose: To understand and apply Memory/CPU Cgroups Limits, Log Rotation Drivers & Isolated Internal Networks in real Docker systems — specifically: Prevent a single runaway conta...

What Topic Is Next → Next Topic 16.3: Container Filesystem/Overlay2 Internals & Docker Troubleshooting Interviews

16.3 Container Filesystem/Overlay2 Internals & Docker Troubleshooting Interviews [65 mins • Assignment (100M) • Hands-On Lab (50M) • 2 Mock Sets • 2 AI Vivas]

Subtopics Covered: Security Hygiene & Data Safety Rules for Container Filesystem/Overlay2 Internals & Docker Troubleshooting Interview... • Performance Optimization, Complexity & Latency Impact of Container Filesystem/Overlay2 Internals & Docker Troublesh... • Live Debugging, Stack • Runnable Code Implementation (DOCKERFILE) • Hands • Idiomatic DOCK Implementation: multi-stage BuildKit Dockerfiles, distroless/Alpine images & Compose networks • Linux Namespaces, cgroups v2, OverlayFS Layers & containerd/runc OCI Execution & Memory Lifecycle for Container Filesystem/Overlay2 Internals • Production Hardening & Security: non-root USER, read-only rootfs, dropped capabilities & Trivy CVE gates

Purpose: To understand and apply Container Filesystem/Overlay2 Internals & Docker Troubleshooting Interviews in real Docker systems — specifically: Inspect namespaces, cgroups, zo...

What Topic Is Next → Module 16 Capstone → End-of-Course Live Client Project: ContainerForge Multi-Service Production Stack & Hardened Image Supply Chain

Module 16 Hands-On Capstone Project (100 Marks): Health-Gated Reverse-Proxy + App + DB Compose Stack with Log/Memory Limits — Architect, implement, test, and document 'Health-Gated Reverse-Proxy + App + DB Compose Stack with Log/Memory Limits' applying every bullet-wise engineering concept from Module 16 (Productio...

3. LIVE CLIENT PRODUCTION PROJECT & CAREER SCOPE

End-of-Course Live Client Production Project + Official Internship Confirmation Letter

Project: ContainerForge Multi-Service Production Stack & Hardened Image Supply Chain • **Confirmed Role:** Docker & Container Engineering Intern

Domain & Production Infra: SaaS Platform DevOps Modernization Client (Multi-Stage Dockerfiles + Docker Compose Production + Private Registry + TLS)

- Engineered minimal multi-stage Dockerfiles running as non-root users, cutting production image size by over 65%.
- Orchestrated isolated bridge networks, named persistent volumes, secrets, and health-checked services via Compose.
- Integrated automated image vulnerability scanning, layer caching, and tagged registry releases in CI/CD.
- Deployed the containerized multi-service application stack to a live HTTPS client production host.

Target Job Roles: Docker & Containerization Engineer, Associate DevOps & Cloud Engineer, Build & Release Automation Engineer, Container-Ready Backend Developer

Realistic Annual CTC Scope: **Fresher / Entry-Level DevOps & Cloud (0–1 Yr):** ₹5.0 LPA – ₹9.5 LPA **Mid-Level Container & DevOps Engineer (1–3 Yrs):** ₹10.5 LPA – ₹20.5 LPA **Senior Cloud-Native / Platform Engineer (3–6+ Yrs):** ₹21.0 LPA – ₹40.0+ LPA