Exercise 8

1.
The main difference between a Docker container and a virtual machine (VM) lies in how they virtualize resources and what layers of the system they encapsulate:

1. Architecture

- **Docker Container**:
    - Shares the host OS kernel
    - Runs isolated processes within the host OS
    - Uses Docker Engine to manage containers
- **Virtual Machine**:
    - Emulates entire hardware through a hypervisor
    - Each VM runs its own OS (guest OS)
    - Managed by a hypervisor (e.g., VMware, VirtualBox, Hyper-V)

2. Performance and Efficiency

- **Docker Container**:
    - Lightweight and fast to start
    - Lower overhead (no guest OS)
    - Better suited for microservices and CI/CD pipelines
- **Virtual Machine**:
    - Heavier and slower to boot
    - Requires more memory and storage
    - Useful when you need full OS isolation or to run different OS types

3. Isolation

- **Docker Container**:
    - Isolated at the process level
    - Less secure by default (shared kernel)
- **Virtual Machine**:
    - Isolated at the hardware level
    - Stronger security boundaries, suitable for untrusted workloads

4. Use Cases

- **Docker**:
    - Deploying microservices

    - Building and testing software quickly
    - Running lightweight, consistent environments
- **VMs**:
    - Running multiple OSes on one machine
    - Hosting legacy or heavy applications
    - Scenarios needing full OS-level isolation

2.

The main purpose of CI is to frequently Integrate code into a shared repository and run automated tests.

4.

Tool	Primary Use
Jenkins	CI/CD pipeline orchestration – Automates build, test, and deployment
Kubernetes	Container orchestration – Manages containerized applications in production
Ansible	Configuration management and automation – Installs packages, configures servers
Terraform	Infrastructure as code – Provisions and manages cloud infrastructure

Jenkins

- Automates the entire software delivery process (build → test → deploy).
- Highly extensible with plugins for Git, Docker, Kubernetes, etc.
- Supports integration with tools like Ansible, Terraform, and even Kubernetes.

So while Kubernetes, Ansible, and Terraform can support parts of the pipeline (e.g., deploy infrastructure or manage environments), Jenkins is the central tool typically used to orchestrate CI/CD pipelines themselves.

5.

1. **Parallel Testing**

- Split tests into groups and run them in parallel across multiple executors/agents.
- Most CI tools (like Jenkins, GitHub Actions, GitLab CI) support this.
- Can drastically reduce runtime if your tests are numerous and independent.

2. Test Selection (Test Impact Analysis)

- Run only the tests affected by the code changes.

- Tools like diff-cover, Codecov, or built-in Git diffs can help.
- Great for large test suites where most tests are irrelevant to small changes.

3. Test Caching

- Use build and test caching to avoid re-running expensive tests or builds.
- Cache dependencies (e.g., `npm`, `pip`, `maven`, Docker layers).
- Reuse previous successful test results if inputs haven't changed.

4. Optimize Tests

- Refactor slow or flaky tests.
- Avoid unnecessary setup/teardown, database access, or external service calls.
- Use mocking/stubbing instead of full system integration where possible.

5. Use Faster Runners

- Upgrade your CI runners/executors to faster hardware or cloud instances.
- Use dedicated runners for critical stages.

6. Split by Test Type

- Separate unit, integration, and end-to-end (E2E) tests into distinct stages.
- Run unit tests first (fastest feedback).
- Delay heavier tests (e.g., E2E) until later or trigger them conditionally.

7. Fail Fast

- Configure the pipeline to stop early when critical tests fail.
- Avoid wasting time running the rest of the pipeline unnecessarily.

8. Test in Containers

- Run tests in pre-built Docker containers to ensure consistent and fast setup.
- Reduces environment setup time.