Why Containers?

Containers are a lightweight, portable, and consistent way to package, distribute, and run applications. They address several challenges in software development and deployment:

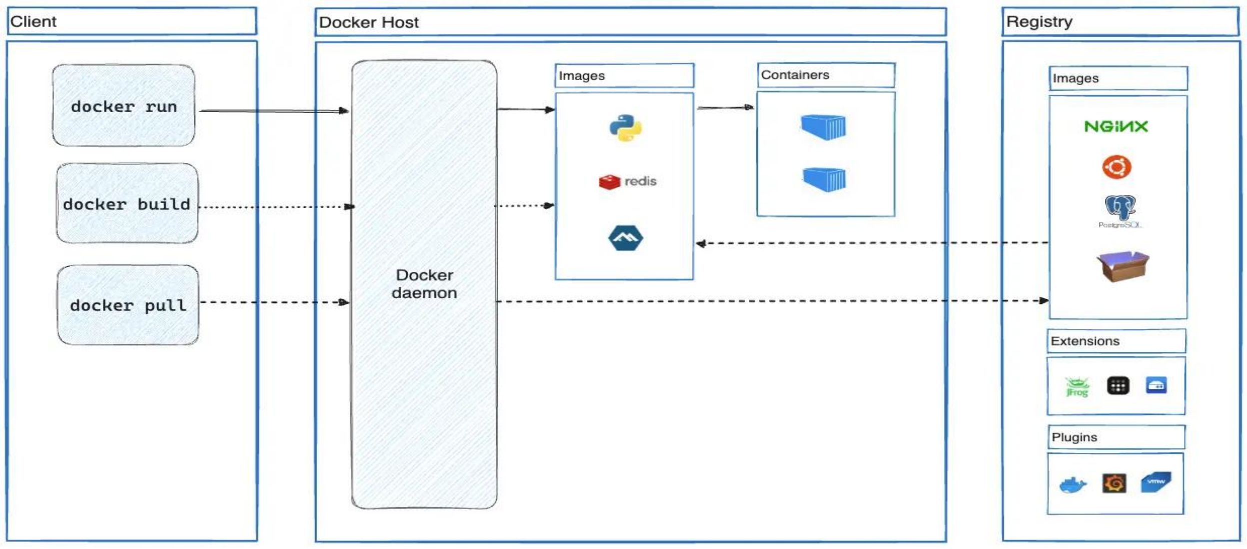
- **Consistency Across Environments**: Containers encapsulate an application with its dependencies (libraries, runtime, configuration), ensuring it runs the same way in development, testing, and production.
- **Portability**: Containers can run on any system with a compatible container runtime (e.g., Docker), regardless of the underlying OS or hardware.
- **Resource Efficiency**: Unlike virtual machines, containers share the host OS kernel, making them lightweight and fast to start.
- **Isolation**: Containers provide process and resource isolation, preventing conflicts between applications.
- **Scalability**: Containers are easy to replicate and orchestrate, making them ideal for microservices and cloud-native applications.
- **Simplified Deployment**: Containers streamline deployment by bundling everything an application needs, reducing "it works on my machine" issues.

What is Docker?

Docker is an open-source platform that automates the deployment of applications inside containers. It simplifies the creation, distribution, and execution of containers, making it easier to manage applications across different environments.

Docker uses containerization technology to package applications and their dependencies into standardized units called **Docker containers**.

Components of Docker



1. Docker Container

- A Docker container is a lightweight, standalone, executable package that includes everything needed to run an application: code, runtime, libraries, and configuration.
- **Characteristics**:
  - Runs as an isolated process on the host OS.
  - Created from a Docker image.
  - Multiple containers can run simultaneously, each isolated from others.
- **Example Use**: Running a web server (e.g., Nginx) in one container and a database (e.g., MySQL) in another, both on the same host.

2. Docker Client

- The Docker client is the primary interface for users to interact with Docker, typically through a command-line interface (CLI) or API.
- **Function**:
  - Sends commands (e.g., docker run, docker build) to the Docker daemon.
  - Can communicate with the daemon on the same host or remotely.
- **Example**: Running docker run -it ubuntu bash to start an interactive Ubuntu container.

3. Docker Daemon

- The Docker daemon (dockerd) is a background service that manages Docker objects like containers, images, networks, and volumes.

- **Function**:
    - Listens for Docker client requests.
    - Handles container lifecycle (create, start, stop, delete), image management, and networking.
    - Runs on the host machine and interacts with the OS kernel.
- **Example**: When you run docker pull, the daemon downloads the image from a registry.

4. Docker Image

- A Docker image is a read-only template used to create containers. It contains the application, its dependencies, and configuration instructions.
- **Characteristics**:
    - Built using a Dockerfile, which defines the steps to set up the environment.
    - Stored as a series of layers, each representing a change (e.g., installing a package).
    - Immutable; changes to a running container create a new image if committed.
- **Example**: The nginx:latest image contains the Nginx web server and its dependencies.

5. Docker Registry

- A Docker registry is a repository for storing and distributing Docker images.
- **Function**:
    - Hosts images, allowing users to push (upload) and pull (download) them.
    - Public registries (e.g., Docker Hub) host shared images; private registries can be set up for internal use.
    - Supports versioning (e.g., nginx:1.21 vs. nginx:latest).
- **Example**: Running docker pull mysql:8.0 downloads the MySQL 8.0 image from Docker Hub.

How Docker Works?

Docker leverages OS-level virtualization to create and manage containers.

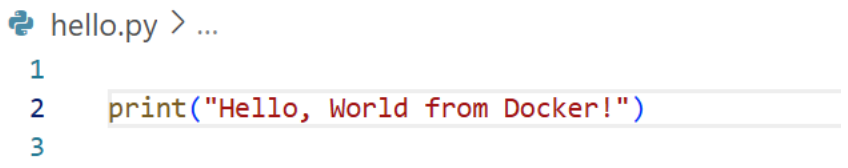
1. **Docker Image Creation**: A developer creates a **Docker image** (a read-only template) using a Dockerfile, which specifies the application, its dependencies, and configuration.
2. **Container Creation**: The Docker engine uses the image to create a **Docker container**, an isolated, runnable instance of the image.
3. **Container Execution**: The container runs on the host OS, sharing its kernel but isolating the application's processes, filesystem, and network.
4. **Docker Daemon**: The **Docker daemon** (dockerd) manages containers, images, networks, and storage on the host system.
5. **Docker Client**: Users interact with Docker via the **Docker client** (CLI or API), sending commands to the daemon to build, run, or manage containers.
6. **Docker Registry**: Images are stored in and pulled from a **Docker registry** (e.g., Docker Hub), a repository for sharing and distributing images.

Docker uses the host OS's kernel features to provide isolation and resource control, ensuring containers are lightweight and efficient.

Steps to Create an Image and Run a Container for helloworld.py

1. **Create your Python script**

   Save the following code as helloworld.py:



hello.py > ...

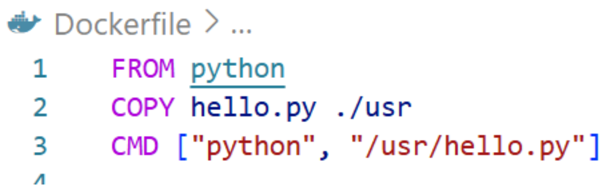
```python

print("Hello, World from Docker!")

```

2. **Write a Dockerfile**

   Create a file named Dockerfile (no extension) in the same folder:



Dockerfile > ...

```dockerfile
FROM python
COPY hello.py ./usr
CMD ["python", "/usr/hello.py"]

```

3. **Build the Docker image**

   In the terminal, navigate to the folder containing the Dockerfile and run:

   ```
   docker build -t helloworld-image .
   ```

4. Run a container from the image

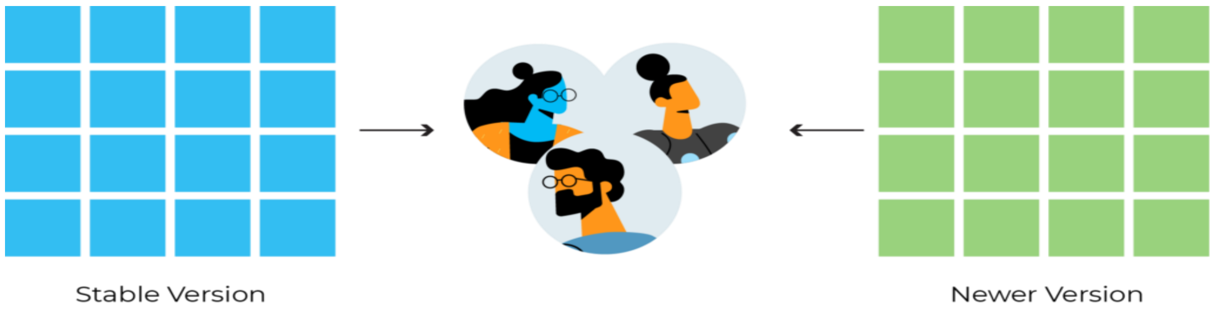
   ```
   docker run --name helloworld-container helloworld-image
   ```

<u>DEPLOYMENT STRATEGIES</u>

A deployment strategy is any technique employed by DevOps teams to successfully launch a new version of the software solution they provide.

Types of Deployment Strategies

1. Blue-Green Deployment



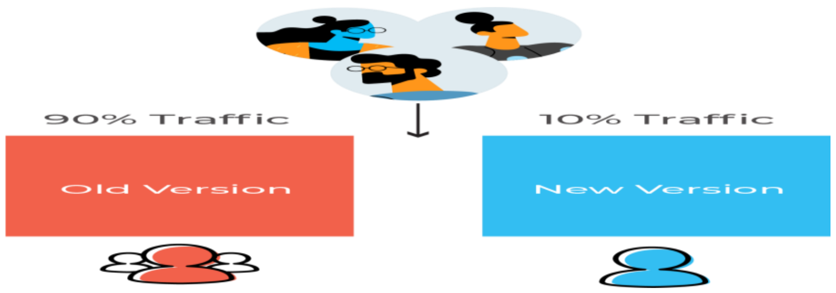
Process: Run two environments (Blue = current, Green = new). Switch traffic to Green once it's ready.

Pros: Near-zero downtime, easy rollback.

Cons: Requires double infrastructure temporarily.

Use case: Critical apps where downtime is not acceptable.

2. Canary Deployment



3.

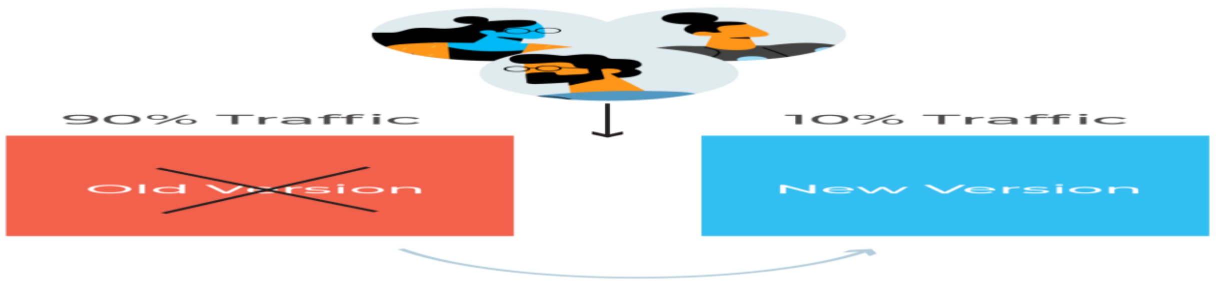
Process: Release the new version to a small subset of users, then gradually increase coverage.

Pros: Reduces risk, allows real-world testing.

Cons: Monitoring is essential; rollout can be slower.

Use case: Large-scale systems, user-facing apps.

3. Recreate Deployment



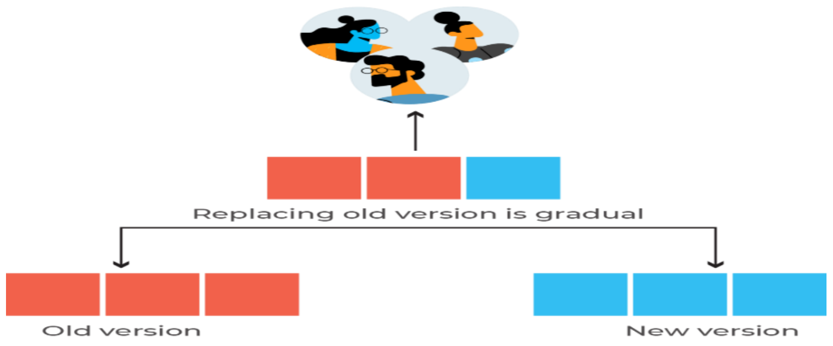
Process: Shut down the old version completely, then start the new one.

Pros: Simple and easy.

Cons: Causes downtime.

Use case: Small apps, non-critical systems.

4. Ramped (Rolling) Deployment



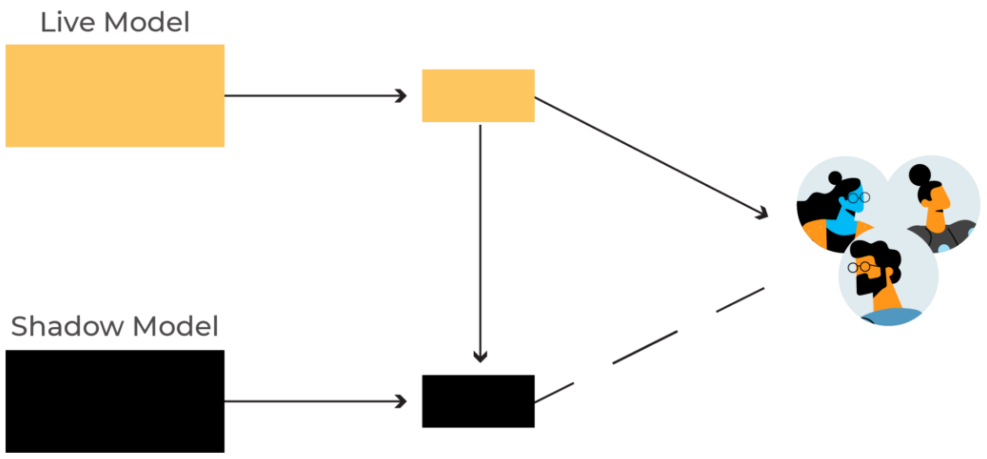
Process: Gradually replace old instances with new ones, one batch at a time.

Pros: No downtime, smooth transition.

Cons: Rollback can be complex.

Use case: Most modern cloud-native apps.

5. Shadow Deployment (a.k.a. Dark Launch)



Process: New version receives real traffic in parallel (copy of production traffic), but responses are ignored by users.

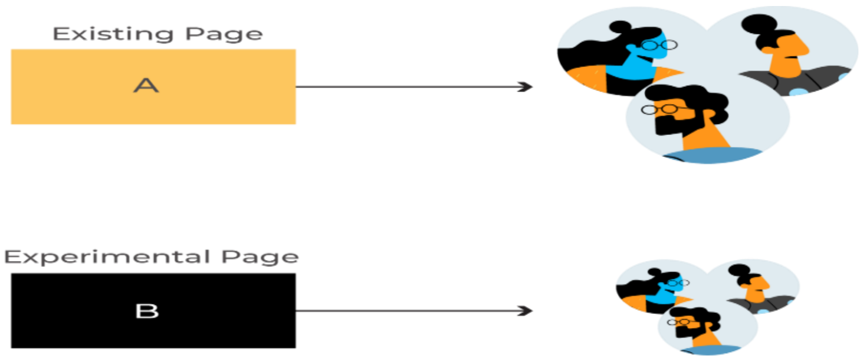
Pros: Test performance under real load without user impact.

Cons: Resource-intensive, tricky to manage.

Use case: Testing experimental features, validating performance.

A/B Testing Deployment

A/B Testing Deployment



Process: Route a percentage of users to version A (old) and version B (new). Compare results.

Pros: Measures impact on user behavior.

Cons: Requires analytics and user segmentation.

Use case: UI/UX changes, feature validation.

Strategy	Downtime	Rollback	Infra Cost	Risk Level	Use Case

Recreate	High	Medium	Low	High	Small apps
Rolling	Low	Medium	Medium	Medium	Web apps with replicas
Blue-Green	Very Low	Easy	High	Low	Critical apps needing fast rollback
Canary	Very Low	Easy	Medium	Low	Large-scale gradual rollout
Shadow	None	N/A	High	Very Low	Performance/load validation
A/B Testing	None	Medium	High	Low	UX/feature experimentation