# [Dockerfiles: Best Practices Notes](https://testdriven.io/blog/docker-best-practices/" \l "bonus-tips)

**Use Multi-stage Builds**

Separate the build environment from the final runtime environment in your Dockerfile. Use a "builder" stage with all the necessary tools to compile/prepare your app, then copy *only* the finished files to a clean, minimal final stage. This creates dramatically **smaller and more secure** production images by discarding all build-time dependencies.

**Order Dockerfile Commands Appropriately**

Structure your Dockerfile from least to most frequently changed commands to maximize Docker's build cache. Place instructions like pip install (which rarely changes) *before* you COPY your source code (which changes often). This makes your rebuilds during development significantly **faster**.

**Use Small Docker Base Images**

Start with the smallest possible base image that fits your needs, like python:3.12-slim instead of the full python:3.12. Smaller base images lead to smaller final images, which means **faster deployments** and a **reduced security attack surface**.

**Minimize the Number of Layers**

Combine related RUN commands using &&. This is most important for cleaning up temporary files (like apt-get caches) in the same layer they were created. This prevents temporary data from being stored in the final image, effectively **reducing its size**.

**Use Unprivileged Containers**

Create a dedicated non-root user in your Dockerfile and switch to it with the USER instruction before running your application. This is a critical security practice that follows the principle of least privilege, **limiting the damage** an attacker could do if they compromise your container.

**Prefer COPY Over ADD**

Use COPY for copying local files and directories into your image. It is more explicit and predictable. ADD has extra features (like unpacking archives and downloading from URLs) that can be unexpected. Stick to COPY for simplicity and clarity.

**Cache Python Packages to the Docker Host**

Use the --mount=type=cache flag on your pip install command to store downloaded packages in a cache on the host machine. This **dramatically speeds up builds** by preventing pip from re-downloading dependencies every time.

**Run Only One Process Per Container**

Each container should have a single responsibility (e.g., one container for your web app, another for your database). This makes your application easier to **scale, monitor, and debug**. It also ensures that container health checks are reliable. This is why tools like Docker Compose are used to manage multi-container applications.

**Prefer Array Over String Syntax**

Use the "exec" form (array syntax like ["python3", "app.py"]) for CMD and ENTRYPOINT. This runs the command directly without a shell, which is safer and prevents issues with signal handling. The string syntax ("python3 app.py") can cause unexpected behavior because it's processed by a shell.

**Understand the Difference Between ENTRYPOINT and CMD**

Use ENTRYPOINT to set the main, fixed executable for the container (e.g., ["python3"]). Use CMD to provide the default, overridable arguments for that executable (e.g., ["app.py"]). When used together, they create a container that acts like a command-line tool.

**Include a HEALTHCHECK Instruction**

Define a HEALTHCHECK command in your Dockerfile to test if your application is truly working, not just running. For a web server, this could be a curl command to a status endpoint. This allows Docker and orchestrators to **automatically restart unhealthy containers**, improving application reliability.

**Bonus Tips**

**Using Python Virtual Environments**

It's still a good practice to use a virtual environment inside a container. It helps **isolate your application's dependencies** from system-level Python packages, preventing potential conflicts and ensuring a clean, predictable environment.

**Set Memory and CPU Limits**

In production, always set resource limits for your containers. This prevents a single container from consuming all the host's memory or CPU, which could crash other containers or the entire server. It ensures **stable and predictable performance**.

**Log to stdout or stderr**

Write your application's logs directly to the standard output (stdout) and standard error (stderr) streams instead of to files. This allows the container platform (like Docker or Kubernetes) to **centrally collect, manage, and route your logs** to other services.

**Use a Shared Memory Mount for Gunicorn Heartbeat**

If you use the Gunicorn web server, mount a shared memory volume by adding the --shm-size flag to your docker run command. This prevents issues where Gunicorn workers might time out and restart unnecessarily, **improving the stability** of your Python web application.

**Secure Communication with TLS**

Protect the Docker daemon socket by configuring it to use TLS encryption. This is crucial if you need to access the Docker daemon over a network, as it **prevents unauthorized access** and ensures all communication is secure.