Dockerfile CMD

Feeling confused is totally fair — Docker's CMD has a few forms that look similar but behave differently. Here's a clean, complete guide you can rely on.

What CMD does

- **Purpose:** Sets the default command (and/or default arguments) that runs when the container starts.
- **Count: Only the last CMD** in a Dockerfile takes effect.
- **Override:** `docker run <image> <command>` overrides CMD completely, unless you combine it with `ENTRYPOINT` in a specific way.
- **Relation to ENTRYPOINT:** CMD can either be the whole command, or just the default arguments for `ENTRYPOINT`.

The three forms of CMD

Shell form

```
CMD echo "Hello from shell"
```

- **How it runs:**
  Command: `/bin/sh -c "echo \"Hello from shell\""`
- **Pros:** Supports shell features (globbing, `&&`, pipes, variable expansion).
- **Cons:** Shell wraps the process, which can affect signal handling and PID 1 behavior.
- **Use when:** You need shell parsing or compound commands.

Exec (JSON array) form

```
CMD ["echo", "Hello from exec"]
```

- **How it runs:**
  Syscall: `execve("echo", ["echo", "Hello from exec"])`

- **Pros:** No shell; exact arguments, better signal handling; recommended best practice.
- **Cons:** No shell features unless you explicitly call a shell: `["/bin/sh", "-c", "echo hi"]`.
- **Use when:** You want predictability, clean signals, explicit args.

Parameter form (with ENTRYPOINT)

```
ENTRYPOINT ["python"]
CMD ["app.py", "--port", "8080"]
```

- **How it runs:**
  `python app.py --port 8080`
- **Purpose:** CMD provides default arguments to ENTRYPOINT.
- **Override behavior:**
  - `docker run image` → uses default args from CMD.
  - `docker run image other.py` → replaces CMD args: `python other.py`.
  - `docker run --entrypoint /bin/bash image` → replaces ENTRYPOINT entirely.

Common patterns and pitfalls

Keeping a container alive

- **Exec form (preferred):**

  ```
  CMD ["tail", "-f", "/dev/null"]
  ```

- **Shell form (works):**

  ```
  CMD tail -f /dev/null
  ```

- **Alternative:**

  ```
  CMD ["sleep", "infinity"]
  ```

Correct argument splitting

- **Incorrect (single arg combines flag+path):**

  ```
  CMD ["tail", "-f /dev/null"]
  ```

- **Correct (each arg separate):**

  ```
  CMD ["tail", "-f", "/dev/null"]
  ```

Environment variables

- **Shell form expands:**

  ```
  ENV PORT=8080
  CMD echo "Port is $PORT"
  ```

- **Exec form does not expand unless a shell is invoked:**

  ```
  ENV PORT=8080
  CMD ["sh", "-c", "echo Port is $PORT"]
  ```

Multiple CMDs

- **Only last one takes effect:**

  ```
  CMD ["echo", "first"]
  CMD ["echo", "second"]  # This is the one that runs.
  ```

How CMD and ENTRYPOINT interact

Quick comparison

Scenario	ENTRYPOINT	CMD	docker run behavior
CMD only	—	Command	`docker run image` runs CMD; extra args replace CMD entirely
ENTRYPOINT (exec) + CMD (args)	Program	Default args	`docker run image` uses default args; extra args replace CMD but keep ENTRYPOINT
Override ENTRYPOINT	Program	Default args	`docker run --entrypoint /bin/bash image` ignores CMD unless passed as args to bash

> Sources: not applicable

Examples

- **Default args pattern:**

  ```
  ENTRYPOINT ["nginx"]
  CMD ["-g", "daemon off;"]
  ```

  - `docker run image` → `nginx -g 'daemon off;'`
  - `docker run image -g 'daemon on;'` → overrides CMD args.

- **Full command via CMD (no ENTRYPOINT):**

  ```
  CMD ["gunicorn", "app:app", "--bind", "0.0.0.0:8000"]
  ```

  - `docker run image` → runs gunicorn.
  - `docker run image bash` → runs `bash` instead (CMD replaced).

Best practices you can trust

- **Prefer exec form:** Precise args, better signal handling, fewer quoting issues.
- **Use ENTRYPOINT for the main binary, CMD for default args:** Makes overrides ergonomic.
- **Avoid shell unless needed:** If you need pipes/ `&&` , use `sh -c` explicitly in exec form.
- **Keep CMD simple and single-purpose:** Complex startup logic belongs in an entrypoint script.
- **Log and exit on failure:** Don't hide errors with `tail -f` unless your goal is a passive base.

Quick reference cheat sheet

- **Run a single process cleanly:**

  ```
  CMD ["python", "server.py"]
  ```

- **Provide default args to a fixed binary:**

  ```
  ENTRYPOINT ["python", "server.py"]
  CMD ["--port", "8080"]
  ```

- **Use shell features:**

```
CMD ["sh", "-c", "python server.py && echo started"]
```

- **Keep container idle:**

```
CMD ["sleep", "infinity"]
```

If you want, I can tailor these patterns to your current stack (Flask, Gunicorn, Kubernetes) and show how overrides behave with `docker run` and `kubectl` .

Perfect — let's dive into **ENTRYPOINT** in Dockerfiles, because it often gets confused with `CMD`.

🐳 ENTRYPOINT Overview

- **Purpose:** Defines the *main executable* that will always run when the container starts.
- **Difference from CMD:**
  - `CMD` = default command or default arguments.
  - `ENTRYPOINT` = fixed command (cannot be overridden unless you explicitly use `--entrypoint` in `docker run`).

Think of `ENTRYPOINT` as the "program" and `CMD` as the "default options" for that program.

🔑 Forms of ENTRYPOINT

1. Exec form (recommended)

```
ENTRYPOINT ["python", "app.py"]
```

- Runs directly as `execve("python", ["python", "app.py"])`.
- No shell involved → clean signal handling.
- Best practice.

2. Shell form

```
ENTRYPOINT python app.py
```

- Runs as `/bin/sh -c "python app.py"`.
- Allows shell features (pipes, env expansion).
- Less predictable for signals.

🧩 ENTRYPOINT + CMD Together

This is where the magic happens:

Example 1: ENTRYPOINT with default args

```
ENTRYPOINT ["python", "app.py"]
CMD ["--port", "8080"]
```

- Default run:

  ```
  docker run image
  # → python app.py --port 8080
  ```

- Override CMD args:

  ```
  docker run image --port 9090
  # → python app.py --port 9090
  ```

- Override ENTRYPOINT:

  ```
  docker run --entrypoint bash image
  # → bash
  ```

Example 2: ENTRYPOINT as fixed binary

```
ENTRYPOINT ["nginx"]
CMD ["-g", "daemon off;"]
```

- Default run:

  ```
  docker run image
  # → nginx -g 'daemon off;'
  ```

- Override CMD args:

  ```
  docker run image -g 'daemon on;'
  # → nginx -g 'daemon on;'
  ```

⚡ Key Differences Between CMD and ENTRYPOINT

Feature	CMD	ENTRYPOINT
Purpose	Default command or args	Fixed command
Override	Replaced by `docker run <command>`	Only overridden with `--entrypoint`
Best use	Provide defaults	Define the main executable
Signal handling	Better in exec form	Better in exec form

✅ Best Practices

- Use **ENTRYPOINT** for the main binary (e.g., `python` , `nginx` , `gunicorn`).
- Use **CMD** for default arguments (e.g., `app.py` , flags).
- Prefer **exec form** (`["cmd", "arg1", "arg2"]`) for both.
- If you need shell features, wrap explicitly:

```
ENTRYPOINT ["sh", "-c", "echo Hello && exec myapp"]
```

👉 Saarthak, since you're working with Flask and Kubernetes setups, a common pattern is:

```
FROM python:3.11-slim
WORKDIR /app
COPY . .
ENTRYPOINT ["python"]
CMD ["app.py"]
```

This way, `ENTRYPOINT` fixes Python as the runtime, and `CMD` provides the default script.

Would you like me to show you **how Kubernetes handles CMD vs ENTRYPOINT** (because `command` and `args` in a Pod spec map directly to these)? That's often the next confusing step.

Perfect — let's connect **CMD** and **ENTRYPOINT** directly to how Kubernetes handles them, because this is where most people get tripped up.

🐳 Dockerfile vs Kubernetes Pod Spec

In Docker:

- **ENTRYPOINT** = the fixed executable.
- **CMD** = default arguments (or default command if ENTRYPOINT isn't set).

In Kubernetes:

- Pod spec has two fields:
  - `command` → maps to Docker's **ENTRYPOINT**
  - `args` → maps to Docker's **CMD**

🔑 Mapping Table

Dockerfile	Kubernetes Pod Spec
`ENTRYPOINT ["python", "app.py"]`	`command: ["python", "app.py"]`
`CMD ["--port", "8080"]`	`args: ["--port", "8080"]`
`CMD ["python", "app.py"]` (no ENTRYPOINT)	`command: ["python", "app.py"]`

⚡ Examples

Example 1: ENTRYPOINT + CMD

Dockerfile:

```
ENTRYPOINT ["python", "app.py"]
CMD ["--port", "8080"]
```

Kubernetes Pod:

```
containers:
- name: myapp
  image: myimage
  command: ["python", "app.py"]
  args: ["--port", "8080"]
```

- Default run: `python app.py --port 8080`
- Override args in Pod spec:

  ```
  args: ["--port", "9090"]
  ```

  → `python app.py --port 9090`

Example 2: CMD only

Dockerfile:

```
CMD ["python", "app.py"]
```

Kubernetes Pod:

```
containers:
- name: myapp
  image: myimage
  command: ["python", "app.py"]
```

Here, CMD becomes the `command` field because there's no ENTRYPOINT.

Example 3: ENTRYPOINT only

Dockerfile:

```
ENTRYPOINT ["nginx"]
```

Kubernetes Pod:

```
containers:
- name: web
```

```
  image: nginx:latest
  command: ["nginx"]
```

If you want to pass flags, you'd use `args` in the Pod spec.

✅ Best Practices in Kubernetes

- Use **ENTRYPOINT** in Dockerfile for the main binary.
- Use **CMD** for default args, which Kubernetes can override via `args`.
- In Kubernetes, prefer setting overrides in `args` rather than replacing `command`, unless you truly want to change the binary.
- For complex startup logic, wrap it in an entrypoint script and set that as ENTRYPOINT.

👉 Saarthak, since you're deploying Flask apps and experimenting with Kubernetes clusters, this mapping is crucial. A common pattern for Python apps is:

```
ENTRYPOINT ["python"]
CMD ["app.py"]
```

Then in Kubernetes:

```
containers:
- name: flask
  image: myflask:latest
  command: ["python"]
  args: ["app.py", "--port", "8080"]
```

That way, you can override just the args in your Pod spec without touching the runtime.

Would you like me to also show you **how Helm charts handle command/args overrides**? That's the next level when you start templating deployments.