Docker: why your image rebuild ignores your code change

📅 2026-09-08 📊 intermediate ["docker" "dockerfile" "build" "caching"]

The Problem

You change a line of code, run `docker build`, and watch the `RUN npm install` step take 90 seconds again. Or worse: the build completes, you run the container, and your code change isn't there.

The Mistake

Putting `COPY . .` (or `COPY src/`) at the top of your Dockerfile. Every build then invalidates everything below it because the layer's input (the whole source tree) has a new hash.

The Fix

Copy only the manifest files first, install dependencies, *then* copy the source:

```
# 1. Manifests first — changes only when dependencies change
COPY package.json package-lock.json ./
RUN npm ci

# 2. Source code — changes whenever you edit a file
COPY . .

# 3. Build
RUN npm run build
```

Now `npm ci` only re-runs when `package.json` or `package-lock.json` actually changes. Code edits reuse the cached layer.

Why It Works

Docker builds each instruction into a layer. The layer's cache key is the hash of the instruction plus the hash of its inputs (files added with `COPY`, env vars, etc.). If either changes, the layer is rebuilt — and every layer *after* it must also rebuild because they depend on its filesystem.

By separating "rarely changes" (manifests, base images) from "always changes" (your source), you maximise cache hits.

Copy-Paste

```
# Multi-stage pattern (Node example)
FROM node:20-alpine AS deps
WORKDIR /app
COPY package.json package-lock.json ./
RUN npm ci --omit=dev

FROM node:20-alpine AS runner
WORKDIR /app
COPY --from=deps /app/node_modules ./node_modules
COPY . .
CMD ["node", "server.js"]
```

Next Steps

- Add a `.dockerignore` to keep `.git`, `node_modules`, and `tests/` out of the build context — they invalidate the cache for no reason.
- For monorepos, copy only the workspace's own manifest, not the whole repo.
- Use `docker buildx build --cache-from` in CI to share layers across builds.

Download PDF (1 page)