Optimize for building in the cloud

- Dockerignore files
- Slim base images
- Multi-stage builds
- Fetch remote files in build
- Multi-threaded tools

Docker Build Cloud runs your builds remotely, and not on the machine where you invoke the build. This means that file transfers between the client and builder happen over the network.

Transferring files over the network has a higher latency and lower bandwidth than local transfers. Docker Build Cloud has several features to mitigate this:

- It uses attached storage volumes for build cache, which makes reading and writing cache very fast.
- Loading build results back to the client only pulls the layers that were changed compared to previous builds.

Despite these optimizations, building remotely can still yield slow context transfers and image loads, for large projects or if the network connection is slow. Here are some ways that you can optimize your builds to make the transfer more efficient:

- Dockerignore files
- Slim base images
- Multi-stage builds
- Fetch remote files in build
- Multi-threaded tools

For more information on how to optimize your builds, see Building best practices.

Dockerignore files

Using a `.dockerignore` file, you can be explicit about which local files you don't want to include in the build context. Files caught by the glob patterns you specify in your ignore-file aren't transferred to the remote builder.

Some examples of things you might want to add to your `.dockerignore` file are:

- `.git` — skip sending the version control history in the build context. Note that this means you won't be able to run Git commands in your build steps, such as `git rev-parse`.
- Directories containing build artifacts, such as binaries. Build artifacts created locally during development.
- Vendor directories for package managers, such as `node_modules`.

In general, the contents of your `.dockerignore` file should be similar to what you have in your `.gitignore`.

Slim base images

Selecting smaller images for your `FROM` instructions in your Dockerfile can help reduce the size of the final image. The Alpine image is a good example of a minimal Docker image that provides all of the OS utilities you would expect from a Linux container.

There's also the special `scratch` image, which contains nothing at all. Useful for creating images of statically linked binaries, for example.

Multi-stage builds

Multi-stage builds can make your build run faster, because stages can run in parallel. It can also make your end-result smaller. Write your Dockerfile in such a way that the final runtime stage uses the smallest possible base image, with only the resources that your program requires to run.

It's also possible to copy resources from other images or stages, using the Dockerfile `COPY --from` instruction. This technique can reduce the number of layers, and the size of those layers, in the final stage.

Fetch remote files in build

When possible, you should fetch files from a remote location in the build, rather than bundling the files into the build context. Downloading files on the Docker Build Cloud server directly is better,

because it will likely be faster than transferring the files with the build context.

You can fetch remote files during the build using the Dockerfile `ADD` instruction, or in your `RUN` instructions with tools like `wget` and `rsync`.

Multi-threaded tools

Some tools that you use in your build instructions may not utilize multiple cores by default. One such example is `make` which uses a single thread by default, unless you specify the `make --jobs=<n>` option. For build steps involving such tools, try checking if you can optimize the execution with parallelization.

□□□ #2
□ □□□ □□□ 2026-10-31 19:03:00 UTC
□ □□□ □□□ 2026-10-31 19:03:01 UTC