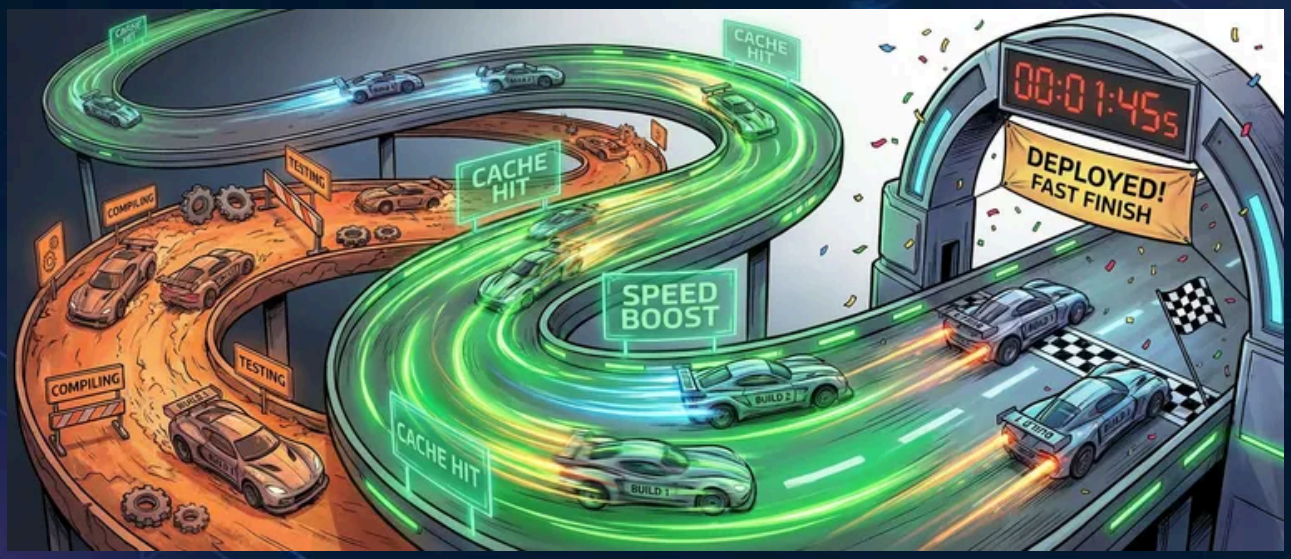
CI Pipeline Caching: Strategies That Actually Work



Published on September 6, 2025

Webstack Builders

Table of Contents

Introduction

CI caching is one of those optimizations that looks straightforward until it isn't. The pitch is simple: cache dependencies and build artifacts so you don't re-download and re-compile the same code on every build. Teams routinely cut 45-minute builds down to 8 minutes. That's real productivity — more deploys per day, faster feedback loops, happier engineers.

But here's what nobody mentions in the "speed up your CI" blog posts: incorrect caching produces builds that pass CI and fail in production.

I've seen this pattern more than once. A team aggressively caches everything, build times plummet, everyone celebrates. Three weeks later, a security patch lands in a transitive dependency. The team runs `npm audit fix`, but they're caching `node_modules` with a key that only hashes `package-lock.json`. The lockfile didn't change (the patch was already in the allowed semver range), so the cache hits and the old, vulnerable version keeps getting restored. The vulnerability makes it to production. The rollback takes longer than all the time they saved.

The core tension is this: caching is an optimization that trades correctness guarantees for speed. Every cache hit assumes that the cached artifact is **identical** to what a fresh build would produce. When that assumption is wrong — and it's surprisingly easy to get wrong — you're shipping artifacts that don't match your source code.

> ⚠ **WARNING**
>
> A fast build that produces incorrect artifacts is worse than a slow build. Before optimizing cache hit ratio, ensure your cache keys capture everything that affects build output.

This article covers the fundamentals of CI caching (cache keys, restore keys, and correctness principles), dependency caching across language ecosystems, Docker layer caching and BuildKit features, the pitfalls that turn caching into a footgun, and how to measure whether your caching is actually working.

The Fundamentals of CI Caching

Why CI Builds Are Slow

Before optimizing, it helps to know where the time actually goes. Here's a breakdown of typical CI pipeline phases and their caching potential:

Phase	Typical Time	Why It's Slow	Caching Opportunity
Checkout	1-5 min	Large repos, shallow clone helps	Git LFS cache
Dependency install	2-15 min	Network latency, extraction	High
Docker build	5-30 min	Layer rebuilds, no cache	Very high
Compile	2-20 min	Full rebuild every time	High
Test	5-60 min	Sequential, no parallelization	Medium (test splitting)
Artifact upload	1-5 min	Large files, slow storage	Low

CI pipeline phases with caching potential.

The biggest wins come from dependency installation and Docker builds — these are both slow **and** highly cacheable. A 45-minute build can realistically drop to 12 minutes with proper caching of just these two phases.

Cache Keys: The Foundation

Cache keys are the mechanism that determines when to use cached artifacts versus rebuilding from scratch. A cache key is a string that uniquely identifies a cached artifact. When your build requests a cache, the CI system looks for an exact match on the key. If found, you get a cache hit. If not, you get a miss and rebuild.

The art is in designing keys that change when (and only when) the cached content should change. A good cache key has three components:

A static prefix

For namespacing and manual invalidation (e.g., `v1-deps-*`)

Dynamic components

Captures everything affecting the cached content (e.g., lockfile hash)

Optionally, a version or OS component

For environment-specific caches (e.g., `runner.os`)

Here's how GitHub Actions cache keys work in practice:

.github/workflows/build.yml

```
# GitHub Actions cache key example
# The "v1-" prefix enables manual cache invalidation — bump to "v2-" to force
# all builds to miss, useful when cache contents become corrupted or stale.
- uses: actions/cache@v4
  with:
    path: node_modules
    key: v1-deps-${{ runner.os }}-${{ hashFiles('**/package-lock.json') }}
    restore-keys: |
      v1-deps-${{ runner.os }}-
      v1-deps-
```

GitHub Actions cache key with lockfile hash and restore key fallbacks.

The `hashFiles()` function computes a hash of the specified files — when `package-lock.json` changes, the hash changes, and you get a new cache key. The `restore-keys` provide fallbacks: if the exact key doesn't match, the CI system looks for keys starting with those prefixes (in order). For each prefix, it finds all cache entries that start with that string and returns the most recently created one. This means `v1-deps-Linux-` might match `v1-deps-Linux-abc123` from yesterday's build.

> **INFO**
>
> Restore keys provide fallback options when the exact cache key doesn't match. They enable partial cache hits — using a slightly stale cache is usually faster than rebuilding from scratch.

Cache Correctness Principles

Three principles determine whether a cache is safe to use:

1. **Completeness means the cache key includes everything that affects the output.**
   If your compiled artifact depends on the compiler version, the compiler version must be in the cache key. Miss something, and you'll serve stale artifacts.

2. **Determinism means the same inputs always produce the same outputs.**
   If your build embeds timestamps or downloads "latest" versions of tools, caching becomes unsafe — the cached artifact won't match what a fresh build would produce.

3. **Isolation means caches don't leak between unrelated builds.**
   PR builds shouldn't pollute the main branch cache. Without isolation, one build's experimental changes can corrupt another build's cache.

These three principles are easy to state but surprisingly easy to violate. The table below maps each principle to the rule it enforces and the mistake that breaks it — a useful checklist when something is caching incorrectly.

Principle	Rule	Common Violation
Completeness	Key includes all inputs	Forgetting compiler version in key
Determinism	Same inputs → same outputs	Build embeds current timestamp
Isolation	Caches don't leak across builds	PR pollutes main branch cache

Cache correctness principles and common violations.

Dependency Caching

Dependency caching is usually the first optimization teams implement, and for good reason — it's high impact and relatively low risk. The pattern is the same across ecosystems: cache the package manager's download cache or installed packages, keyed on the lockfile hash.

Node.js / npm / yarn / pnpm

The JavaScript ecosystem offers several caching strategies. The simplest is using the built-in caching in `actions/setup-node` .

Note: The following YAML snippets are step excerpts — they belong inside a `jobs.<job_id>.steps:` block in a complete workflow file.

.github/workflows/build.yml

```yaml
# GitHub Actions - npm with built-in caching
- name: Setup Node.js
  uses: actions/setup-node@v4
  with:
    node-version: '20'
    cache: 'npm'  # Automatically caches ~/.npm

- name: Install dependencies
  run: npm ci
```

GitHub Actions npm caching using built-in setup-node cache.

This caches npm's download cache (`~/.npm`), not `node_modules` . On a cache hit, `npm ci` still runs but skips network requests for packages already in the cache. For most projects, this is sufficient and requires minimal configuration.

For more control — or if you want to skip `npm ci` entirely on cache hits — you can cache `node_modules` directly:

.github/workflows/build.yml

```yaml
# GitHub Actions - cache node_modules directly
- name: Cache node_modules
  uses: actions/cache@v4
  id: npm-cache
  with:
    path: node_modules
    key: ${{ runner.os }}-node-${{ hashFiles('**/package-lock.json') }}
    restore-keys: |
      ${{ runner.os }}-node-

- name: Install dependencies
  if: steps.npm-cache.outputs.cache-hit != 'true'
  run: npm ci
```

GitHub Actions with conditional npm install on cache miss.

The conditional install skips `npm ci` entirely when the cache hits. This is faster but has a tradeoff: if something corrupts the cache, you won't notice until the lockfile changes.

For pnpm, cache the content-addressable store rather than `node_modules`. pnpm hardlinks from the store, so restoring the store is effectively instant:

.github/workflows/build.yml

```yaml
# GitHub Actions - pnpm store caching
- name: Setup pnpm
  uses: pnpm/action-setup@v2
  with:
    version: 8

- name: Get pnpm store directory
  id: pnpm-cache
  shell: bash
  run: echo "STORE_PATH=$(pnpm store path)" >> $GITHUB_OUTPUT

- name: Cache pnpm store
  uses: actions/cache@v4
  with:
```

```
      path: ${{ steps.pnpm-cache.outputs.STORE_PATH }}
      key: ${{ runner.os }}-pnpm-${{ hashFiles('**/pnpm-lock.yaml') }}
      restore-keys: |
        ${{ runner.os }}-pnpm-

  - name: Install dependencies
    run: pnpm install --frozen-lockfile
```

pnpm store caching for near-instant installs.

> ✓ SUCCESS
>
> pnpm's content-addressable store makes it particularly cache-friendly. Cache the store directory, not `node_modules` — pnpm will hardlink from the store instantly.

Python / pip / Poetry

Python caching follows the same pattern. For pip, cache the download cache directory:

.github/workflows/build.yml

```
# GitHub Actions - pip caching
- name: Cache pip
  uses: actions/cache@v4
  with:
    path: ~/.cache/pip
    key: ${{ runner.os }}-pip-${{ hashFiles('**/requirements*.txt') }}
    restore-keys: |
      ${{ runner.os }}-pip-

- name: Install dependencies
  run: pip install -r requirements.txt
```

pip download cache configuration.

For Poetry, cache both the Poetry cache and the virtual environment. Setting `virtualenvs.in-project` keeps the venv in the project directory, making it easier to cache:

.github/workflows/build.yml

```yaml
# GitHub Actions - Poetry caching
- name: Cache Poetry
  uses: actions/cache@v4
  with:
    path: |
      ~/.cache/pypoetry
      .venv
    key: ${{ runner.os }}-poetry-${{ hashFiles('**/poetry.lock') }}
    restore-keys: |
      ${{ runner.os }}-poetry-

- name: Install dependencies
  run: |
    poetry config virtualenvs.in-project true
    poetry install --no-interaction
```

Poetry caching with in-project virtual environment.

Ruby / Bundler

Ruby's Bundler has a straightforward caching pattern. The `setup-ruby` action includes built-in caching that handles the common case:

.github/workflows/build.yml

```yaml
# GitHub Actions - Ruby with built-in Bundler caching
- name: Setup Ruby
  uses: ruby/setup-ruby@v1
  with:
    ruby-version: '3.3'
    bundler-cache: true  # Caches gems automatically, keyed on Gemfile.lock
```

Ruby setup with automatic Bundler caching.

The `bundler-cache: true` option handles everything — it caches the gem installation directory and keys on `Gemfile.lock`. For most Ruby projects, this single option is sufficient.

If you need more control (for example, to cache additional directories or use a custom key), you can configure caching manually:

.github/workflows/build.yml

```yaml
# GitHub Actions - manual Bundler caching
- name: Cache Bundler gems
  uses: actions/cache@v4
  with:
    path: vendor/bundle
    key: ${{ runner.os }}-gems-${{ hashFiles('**/Gemfile.lock') }}
    restore-keys: |
      ${{ runner.os }}-gems-

- name: Install dependencies
  run: |
    bundle config path vendor/bundle
    bundle install --jobs 4 --retry 3
```

Manual Bundler caching with vendor directory.

Setting `bundle config path vendor/bundle` keeps gems in the project directory rather than the system location, making the cache path predictable.

Go Modules

Go has two caches worth preserving: the module cache (`~/go/pkg/mod`) containing downloaded dependencies, and the build cache (`~/.cache/go-build`) containing compiled packages. Caching both dramatically speeds up builds:

.github/workflows/build.yml

```yaml
# GitHub Actions - Go module and build cache
- name: Setup Go
  uses: actions/setup-go@v5
  with:
    go-version: '1.22'
    cache: true  # Caches both mod and build cache automatically
```

Go caching using setup-go built-in cache.

The `setup-go` action handles both caches automatically when `cache: true` is set. It keys on `go.sum` by default.

Rust / Cargo

Rust builds are notoriously slow, making caching critical. Cargo has several cache locations, and the `target/` directory containing compiled artifacts can grow very large:

.github/workflows/build.yml

```yaml
# GitHub Actions - Rust caching with dedicated action
- name: Cache Rust
  uses: Swatinem/rust-cache@v2
  with:
    shared-key: "build"  # Share cache across jobs
```

Rust caching using the Swatinem/rust-cache action.

The `rust-cache` action handles the complexity of Cargo's multiple cache directories and applies sensible defaults. It's significantly better than manually configuring `actions/cache` for Rust projects.

> **⚠ WARNING**
>
> Rust's `target/` directory can grow very large. Consider using `sccache` for distributed compilation caching if your builds are still slow after basic caching.

Docker Layer Caching

Docker layer caching is both the biggest opportunity and the most common source of confusion in CI caching. Understanding how it works — and how it breaks — is essential for fast, correct builds.

How Docker Layer Caching Works

Docker builds images as a stack of layers. Build instructions like FROM, COPY, and RUN each create a new layer. (Metadata instructions like ENV and LABEL modify the image configuration but don't create layers in the same way.) When you rebuild, Docker checks each layer in order: if the instruction and all its inputs are identical to a previous build, Docker reuses the cached layer. If anything differs, Docker rebuilds that layer **and every layer after it**.

That last part is critical. Layer caching is sequential — once a layer is invalidated, everything downstream rebuilds. This creates a domino effect that determines whether your build takes 30 seconds or 10 minutes.

Here's a common mistake:

Dockerfile

```
# Inefficient: any source change invalidates npm ci
FROM node:20-alpine
WORKDIR /app
COPY . .
RUN npm ci
RUN npm run build
```

Inefficient Dockerfile that invalidates dependency cache on any source change.

The problem is `COPY . .` before `npm ci`. Every time any file changes — even a README edit — the COPY layer is invalidated, which invalidates npm ci, which means re-downloading all dependencies. On a project with hundreds of dependencies, that's 5+ minutes wasted.

The fix is layer ordering: copy dependency files first, install dependencies, then copy source:

Dockerfile

```
# Optimized: dependency layer is stable
FROM node:20-alpine
WORKDIR /app

# Copy dependency files first
```

```
COPY package.json package-lock.json ./

# Install dependencies (cached unless package files change)
RUN npm ci

# Copy source after dependencies
COPY . .

# Build (must rebuild when source changes)
RUN npm run build
```

Optimized Dockerfile with dependency layer before source copy.

Now the npm ci layer only rebuilds when package.json or package-lock.json changes. Source code changes only invalidate the final two layers.

Multi-Stage Build Caching

Multi-stage builds separate concerns into distinct build phases, which improves both caching and final image size. Each stage can be cached independently, and only the artifacts you explicitly copy make it to the final image.

Dockerfile

```
# Stage 1: Dependencies (highly cacheable)
FROM node:20-alpine AS deps
WORKDIR /app
COPY package.json package-lock.json ./
RUN npm ci

# Stage 2: Build (depends on source)
FROM node:20-alpine AS builder
WORKDIR /app
COPY --from=deps /app/node_modules ./node_modules
COPY . .
RUN npm run build

# Stage 3: Production (minimal image)
FROM node:20-alpine AS runner
WORKDIR /app
```

```
ENV NODE_ENV=production
COPY --from=deps /app/node_modules ./node_modules
COPY --from=builder /app/dist ./dist
COPY package.json ./
CMD ["node", "dist/index.js"]
```

Multi-stage Dockerfile optimized for layer caching.

The `deps` stage is highly cacheable — it only changes when dependencies change. The `builder` stage rebuilds on source changes but starts from cached dependencies. The `runner` stage produces a minimal production image without build tools or dev dependencies.

> **INFO**
>
> BuildKit is Docker's modern build engine, which became the default builder in Docker Engine 23.0 (February 2023). CI environments typically use Docker Engine rather than Docker Desktop, so that's the relevant version for most readers. BuildKit replaced the legacy "classic" builder and offers significant improvements: parallel build stages, better caching, cache mounts, and secret handling. If you're running Docker 23.0 or later, you're already using BuildKit.

With BuildKit, you can build specific stages directly (useful for debugging or cache warming):

Bash

```
docker build --target deps -t myapp:deps .
```

BuildKit Cache Mounts

Traditional Docker layer caching has a limitation: when a layer is invalidated, its cache is gone. BuildKit cache mounts solve this by providing persistent caches that survive layer invalidation.

Dockerfile

```
# syntax=docker/dockerfile:1.4
FROM node:20-alpine
WORKDIR /app
COPY package.json package-lock.json ./

# Cache mount persists npm's download cache across builds
RUN --mount=type=cache,target=/root/.npm \
    npm ci

COPY . .

# Cache mount for Next.js build cache
RUN --mount=type=cache,target=/app/.next/cache \
    npm run build
```

BuildKit cache mounts for persistent caching across builds.

The `--mount=type=cache` directive creates a cache directory that persists across builds. Even when the layer is invalidated (because package-lock.json changed), the npm download cache is still there — npm only downloads packages that aren't already cached.

This is particularly valuable for large dependency trees or slow package registries. The cache mount pattern works for any package manager: pip, cargo, go mod, and others.

> **INFO**
>
> BuildKit cache mounts persist across builds even when the layer is invalidated. This means your npm/pip cache survives source code changes, unlike traditional layer caching.

Remote Docker Cache

Local Docker caching doesn't help in CI where each build runs on a fresh runner. You need remote caching to share layer caches across builds.

BuildKit supports several cache backends. For GitHub Actions, the simplest is the built-in GHA cache:

.github/workflows/build.yml

```yaml
# GitHub Actions with BuildKit GHA cache
- name: Set up Docker Buildx
  uses: docker/setup-buildx-action@v3

- name: Build and push
  uses: docker/build-push-action@v5
  with:
    context: .
    push: true
    tags: ghcr.io/myorg/myapp:${{ github.sha }}
    cache-from: type=gha
    cache-to: type=gha,mode=max
```

GitHub Actions Docker build with GHA cache backend.

The GHA backend uses GitHub's Actions cache (the same one used by `actions/cache`), so it's free and requires no additional setup. The downside is the 10GB cache limit shared across all workflows in the repo.

For larger projects or self-hosted runners, registry-based caching stores layers in your container registry:

.github/workflows/build.yml

```yaml
# GitHub Actions with registry cache
- name: Build and push
  uses: docker/build-push-action@v5
  with:
    context: .
    push: true
    tags: ghcr.io/myorg/myapp:${{ github.sha }}
    cache-from: type=registry,ref=ghcr.io/myorg/myapp:buildcache
    cache-to: type=registry,ref=ghcr.io/myorg/myapp:buildcache,mode=max
```

GitHub Actions Docker build with registry-based caching.

Registry caching works anywhere (not just GitHub Actions) but adds network latency and storage costs. For AWS environments, S3-based caching offers a good balance of performance and cost.

Common Caching Pitfalls

Caching bugs are particularly nasty because they're often invisible — builds pass, tests pass, but the artifacts are wrong. Here are the four patterns I see most often.

Pitfall 1: Over-Caching (Key Too Broad)

Over-caching happens when your cache key doesn't include something that affects the cached content. The classic example is a static cache key that never changes:

.github/workflows/build.yml

```
# BAD: Cache key never changes
- uses: actions/cache@v4
  with:
    path: node_modules
    key: deps-v1  # This key is static!
```

Over-caching with a static key that never invalidates.

When `package-lock.json` changes, the cache key stays the same, so you keep getting the old dependencies. Security patches don't get applied. New packages don't get installed. Builds pass but artifacts are stale.

The fix is including the lockfile hash in the key:

.github/workflows/build.yml

```
# GOOD: Cache key includes lockfile hash
- uses: actions/cache@v4
  with:
    path: node_modules
    key: deps-${{ hashFiles('**/package-lock.json') }}
```

Correct cache key that invalidates when dependencies change.

Pitfall 2: Under-Caching (Key Too Specific)

The opposite problem: your cache key changes too often, so you never get cache hits. I've seen teams use the commit SHA in cache keys:

.github/workflows/build.yml

```
# BAD: Cache key changes every commit
- uses: actions/cache@v4
  with:
    path: node_modules
    key: deps-${{ github.sha }}  # Different every commit!
```

Under-caching where the key changes on every build.

Every build is a cache miss. You've added caching overhead (checking for cache, uploading cache) without any benefit.

The key insight: cache keys should change when the **cached content** should change, not when the **build** changes. For dependencies, that's the lockfile. For compiled artifacts, that's the source files.

Pitfall 3: Cache Pollution

Cache pollution happens when one build writes incorrect content to a cache that other builds use. The most common cause is shared caches between branches:

.github/workflows/build.yml

```
# BAD: All branches share the same cache
- uses: actions/cache@v4
  with:
    path: node_modules
    key: deps-${{ hashFiles('**/package-lock.json') }}
```

Shared cache that allows branch pollution.

Here's the scenario: a PR modifies a postinstall script that changes what ends up in `node_modules`, but the lockfile hash stays the same. The build runs and writes the modified `node_modules` to the cache. Now every build with that lockfile hash — including main branch builds — restores the PR's experimental changes. Alternatively, a PR uses a restore key fallback that matches a stale cache from an abandoned branch, pulling in outdated or broken dependencies.

The fix is branch isolation with restore key fallbacks:

.github/workflows/build.yml

```yaml
# GOOD: Branch isolation with fallback to main
- uses: actions/cache@v4
  with:
    path: node_modules
    key: deps-${{ github.ref }}-${{ hashFiles('**/package-lock.json') }}
    restore-keys: |
      deps-refs/heads/main-${{ hashFiles('**/package-lock.json') }}
      deps-refs/heads/main-
```

Branch-isolated cache with restore keys for efficiency.

Now each branch writes to its own cache namespace, but PRs can still restore from the main branch cache for efficiency. Main branch builds never read from PR caches.

> **DANGER**
>
> Cache pollution is insidious — builds pass but artifacts are incorrect. Always consider whether PR/branch caches should be isolated from the main branch cache.

Pitfall 4: Non-Deterministic Builds

Caching assumes determinism: the same inputs produce the same outputs. If your build isn't deterministic, caching becomes unsafe.

Common sources of non-determinism:

Floating versions

Using `npm install` instead of `npm ci` allows the package manager to resolve different versions on different runs, even with the same lockfile.

Timestamps

If your build embeds the current timestamp in artifacts, a cached artifact won't match a fresh build. Use `SOURCE_DATE_EPOCH` for reproducible builds.

Parallel race conditions

If multiple jobs write to the same cache simultaneously, you get undefined behavior. Use job-specific keys or configure jobs to only read from the cache.

Network fetches

If your build downloads "latest" versions of tools or resources, different builds get different versions. Pin versions and cache the downloads.

The test for non-determinism: run the same build twice with a clean cache. If the outputs differ (beyond timestamps you can't control), you have a non-determinism bug that will eventually cause caching problems.

Advanced Caching Patterns

Once you have basic caching working, these patterns can push build times even lower.

Cache Warming

The first build of the day is always slow — caches have often expired or been evicted overnight. Cache warming runs a scheduled job that pre-populates caches before engineers start working.

.github/workflows/cache-warming.yml

```yaml
name: Cache Warming

on:
  schedule:
    - cron: '0 6 * * 1-5'  # 6 AM UTC weekdays
  workflow_dispatch:  # Allow manual trigger

jobs:
  warm-caches:
```

```
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4

      - uses: actions/setup-node@v4
        with:
          node-version: '20'
          cache: 'npm'

      - run: npm ci

      - uses: docker/setup-buildx-action@v3

      - name: Warm Docker layer cache
        uses: docker/build-push-action@v5
        with:
          context: .
          push: false
          cache-from: type=gha
          cache-to: type=gha,mode=max
```

Scheduled cache warming workflow that runs before business hours.

The workflow runs at 6 AM on weekdays, installs dependencies (populating the npm cache), and builds the Docker image (populating the layer cache). When engineers start their day and push code, the caches are warm and builds are fast.

Distributed Caching for Self-Hosted Runners

GitHub-hosted runners get fresh caches from GitHub's infrastructure, but self-hosted runners need a shared cache backend. S3 is the most common choice for AWS environments.

For dependency caching, you can use tools like `actions/cache` with S3 backend support, or dedicated solutions like Gradle's build cache or Bazel's remote cache. For Docker layer caching, BuildKit's S3 backend works well:

.github/workflows/build.yml

```
  # Docker build with S3 cache backend
```

```
  - name: Build with S3 cache
    uses: docker/build-push-action@v5
    with:
      context: .
      push: true
      tags: ${{ env.IMAGE_TAG }}
      cache-from: type=s3,region=us-east-1,bucket=ci-cache,name=myapp
      cache-to: type=s3,region=us-east-1,bucket=ci-cache,name=myapp,mode=max
```

Docker build with S3-based distributed cache.

The tradeoff is operational complexity: you need to manage the S3 bucket, IAM permissions, lifecycle policies for cache expiration, and potentially cross-region replication if runners are distributed. For most teams, GitHub's built-in caching is simpler and sufficient.

Incremental Build Caching

Many build tools support incremental compilation, where only changed files are recompiled. To leverage this in CI, you need to cache the build tool's internal state.

For TypeScript, cache the `.tsbuildinfo` file that tracks what's been compiled:

.github/workflows/build.yml

```
# TypeScript incremental build caching
- name: Cache TypeScript build info
  uses: actions/cache@v4
  with:
    path: |
      .tsbuildinfo
      dist/
    key: tsc-${{ runner.os }}-${{ hashFiles('src/**/*.ts', 'tsconfig.json') }}
    restore-keys: |
      tsc-${{ runner.os }}-
```

TypeScript incremental build caching.

For Next.js, cache the `.next/cache` directory:

.github/workflows/build.yml

```
# Next.js build caching
- name: Cache Next.js build
  uses: actions/cache@v4
  with:
    path: .next/cache
    key: nextjs-${{ runner.os }}-${{ hashFiles('**/package-lock.json') }}-${{ hashFiles('src/**') }}
    restore-keys: |
      nextjs-${{ runner.os }}-${{ hashFiles('**/package-lock.json') }}-
      nextjs-${{ runner.os }}-
```

Next.js build cache for faster rebuilds.

> ✓ SUCCESS
>
> Next.js, Webpack, and TypeScript all support incremental builds. Caching their intermediate outputs (`.next/cache` , `.tsbuildinfo`) dramatically speeds up rebuilds when only a few files change.

Measuring Cache Effectiveness

You can't improve what you don't measure. Caching without metrics is guesswork — you won't know if your caches are actually helping or if they've silently broken.

Key Metrics

> **INFO**
>
> GitHub Actions shows cache hit/miss in workflow logs and provides cache management at `Settings → Actions → Caches`. GitLab CI shows cache status in job logs. For deeper analysis, the GitHub CLI (`gh api repos/{owner}/{repo}/actions/cache/usage`) and GitLab API provide programmatic access. Third-party tools like Datadog CI Visibility and BuildPulse aggregate these metrics across repositories.

Four metrics tell you whether caching is working:

Cache hit ratio

Is the percentage of builds that found a usable cache. Calculate it as `cache_hits / (cache_hits + cache_misses)`. A healthy ratio is above 80%. Below 50% for more than a day means something is wrong with your cache key design.

Time saved

Is the cumulative time you didn't spend rebuilding. Track this per workflow and per job to identify which caches are most valuable. If a cache saves 30 seconds but adds 45 seconds of overhead (downloading, extracting, uploading), it's not worth it.

Cache size

Matters because storage isn't free, and most platforms have limits. GitHub Actions has a 10GB limit per repository, shared across all workflows. Track current size and growth rate, and alert before you hit limits.

Build time percentiles

(p50, p95) tell you what engineers actually experience. Compare cached builds to uncached builds to quantify the improvement. If your p95 build time is still 20 minutes, caching isn't solving the right problem.

Debugging Cache Misses

When cache hit ratio drops, you need to diagnose why. Here's the debugging workflow:

Step 1: Check the cache key. Add a step to print the computed cache key:

.github/workflows/build.yml

```
- name: Debug cache key
  run: |
    echo "Cache key: deps-${{ runner.os }}-${{ hashFiles('**/package-lock.json') }}"
    echo "Lockfile hash: ${{ hashFiles('**/package-lock.json') }}"
```

Debug step to inspect computed cache keys.

Step 2: Check cache-hit output. The `actions/cache` action outputs whether the cache was found:

.github/workflows/build.yml

```
- uses: actions/cache@v4
  id: cache
  with:
    path: node_modules
    key: deps-${{ runner.os }}-${{ hashFiles('**/package-lock.json') }}

- name: Cache status
  run: echo "Cache hit: ${{ steps.cache.outputs.cache-hit }}"
```

Checking cache hit status in workflow logs.

Step 3: Inspect cache storage. Use the GitHub CLI to see what caches exist and when they were last used:

cache-analysis.sh

```
#!/bin/bash
# List all caches for this repo with size and last access time
gh api repos/{owner}/{repo}/actions/caches \
  | jq -r '.actions_caches[] | "\(.key): \(.size_in_bytes / 1024 / 1024 | floor)MB, last used: \(.last_accessed_at)"'
```

Script to list cache entries via GitHub API.

Common causes of unexpected misses:

Lockfile changed (intended behavior, but verify the change was intentional)
Cache key includes something volatile (like a timestamp or random value)
Cache was evicted due to size limits or inactivity
Restore keys aren't matching due to prefix mismatch
Different runner OS or architecture than the cached build

Platform-Specific Guides

Different CI platforms have distinct caching models. Here's how to implement the principles we've covered on the two most common platforms.

GitHub Actions

GitHub Actions caches are scoped to the repository and branch, with cache entries from the default branch available to all branches. The platform stores caches in Azure Blob Storage, which means restore times depend on cache size and runner location.

This workflow demonstrates the patterns we've discussed: setup actions with built-in caching, explicit `actions/cache` for build artifacts, and Docker BuildKit with GitHub Actions cache backend.

.github/workflows/build.yml

```yaml
name: Build and Test

on:
  push:
    branches: [main]
```

```
    pull_request:

jobs:
  build:
    runs-on: ubuntu-latest

    steps:
      - uses: actions/checkout@v4

      - uses: actions/setup-node@v4
        with:
          node-version: '20'
          cache: 'npm'

      - run: npm ci
```

The `setup-node` action handles dependency caching automatically. It hashes `package-lock.json` and restores `~/.npm` on cache hits. This covers the common case without any additional configuration.

For build artifacts that survive across runs, use `actions/cache` explicitly:

.github/workflows/build.yml

```
      - uses: actions/cache@v4
        with:
          path: |
            .next/cache
            node_modules/.cache
          key: build-${{ runner.os }}-${{ hashFiles('package-lock.json') }}-${{ hashFiles('src/**') }}
          restore-keys: |
            build-${{ runner.os }}-${{ hashFiles('package-lock.json') }}-
            build-${{ runner.os }}-

      - run: npm run build
      - run: npm test
```

The key includes both the lockfile hash (dependencies) and source hash (application code). The restore keys provide fallback: if the exact source changes, we can still reuse the build cache from the same dependencies. Next.js, Webpack, and other bundlers can leverage the partial cache to skip unchanged modules.

For Docker builds, use BuildKit's GitHub Actions cache backend:

.github/workflows/build.yml

```
      - uses: docker/setup-buildx-action@v3

      - uses: docker/build-push-action@v5
        with:
          context: .
          cache-from: type=gha
          cache-to: type=gha,mode=max
          push: ${{ github.event_name != 'pull_request' }}
          tags: myapp:${{ github.sha }}
```

The `type=gha` cache backend stores layer cache in the same storage as `actions/cache`, but it's optimized for Docker's layer-based model. The `mode=max` option caches all layers, not just the final image layers, which is important for multi-stage builds.

GitLab CI

GitLab CI uses a different caching model. Caches are stored either on the runner (for shell and Docker executors) or in object storage (for Kubernetes executors). Cache scope defaults to project and branch, but you can configure it.

The key difference from GitHub Actions: GitLab separates **caches** (for reuse across pipelines) from **artifacts** (for passing data between jobs in the same pipeline). Understanding this distinction is critical.

.gitlab-ci.yml

```
stages:
  - install
  - build
  - test

variables:
  npm_config_cache: "$CI_PROJECT_DIR/.npm"
```

The `npm_config_cache` variable tells npm to store its cache inside the project directory, where GitLab can persist it.

GitLab supports YAML anchors for cache configuration reuse:

.gitlab-ci.yml

```yaml
.node_cache: &node_cache
  key:
    files:
      - package-lock.json
  paths:
    - .npm/
  policy: pull
```

The leading dot in `.node_cache` makes this a "hidden job" in GitLab — it won't execute on its own but serves as a reusable template. The `&node_cache` creates a YAML anchor that other jobs can reference with `<<: *node_cache`. The `files` key syntax hashes the specified files automatically. The `policy: pull` means this job only reads the cache; it doesn't write. This prevents cache corruption from parallel jobs.

.gitlab-ci.yml

```yaml
install:
  stage: install
  cache:
    <<: *node_cache
    policy: pull-push
  script:
    - npm ci --cache .npm --prefer-offline
  artifacts:
    paths:
      - node_modules/
    expire_in: 1 hour
```

The install job uses `policy: pull-push` because it's the only job that should write the cache. It also produces `node_modules/` as an artifact for downstream jobs. Artifacts are guaranteed delivery; caches are best-effort.

.gitlab-ci.yml

```yaml
build:
  stage: build
  cache:
    <<: *node_cache
  dependencies:
    - install
  script:
    - npm run build
  artifacts:
    paths:
      - dist/
    expire_in: 1 week

test:
  stage: test
  cache:
    <<: *node_cache
  dependencies:
    - install
  script:
    - npm test
```

The `dependencies` key tells GitLab to download artifacts from the install job. Combined with the npm cache, downstream jobs have everything they need without re-running `npm ci`.

> **INFO**
>
> **GitLab cache gotcha**: If you're using Kubernetes executors with distributed runners, cache uploads and downloads add latency. For small caches, the transfer time can exceed the time saved. Profile your specific workload to find the break-even point; for npm caches under 50MB, local `npm ci` is often faster than cache restore.

Conclusion

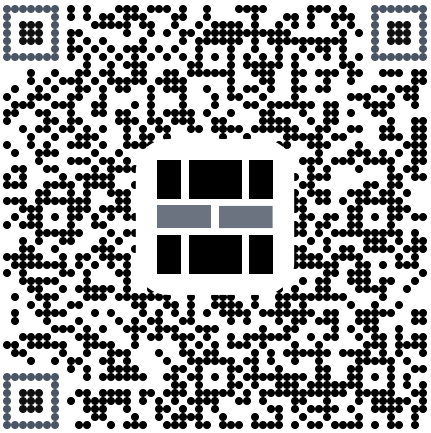
CI caching is deceptively simple on the surface: save files, restore files, skip work. But effective caching requires understanding the principles beneath the configuration.

- **Cache keys must capture everything that affects build output.**
  If your key doesn't include all the inputs, you'll restore stale artifacts. If your key includes too much, you'll never hit the cache. The sweet spot is a key that changes exactly when the cached content should change.
- **Correctness comes before speed.**
  A cache that serves wrong artifacts is worse than no cache. When debugging mysterious failures, always test with a clean cache first. If the problem disappears, your cache key is incomplete.
- **Measure continuously.**
  Cache hit ratio tells you whether caching is working. A 50% hit ratio means half your builds are paying full price. Investigate misses systematically: is the key too specific? Is something non-deterministic? Are caches being evicted?
- **Layer ordering matters for Docker.**
  Put stable content early in your Dockerfile, volatile content late. A single misplaced `COPY . .` can invalidate the entire layer cache on every commit.

The patterns in this article can reduce build times by 70-90%. But the improvement isn't automatic. You need to design your cache keys deliberately, monitor hit ratios continuously, and debug misses when they occur. Fast builds that produce correct artifacts are the goal. Get there systematically.

Once you've optimized caching, the next bottlenecks are usually test execution time (addressed through parallelization and test splitting) and monorepo builds (which benefit from affected-file detection and incremental builds). Caching is the foundation — get it right first, then layer on these more advanced optimizations.

https://www.webstackbuilders.com/articles/ci-pipeline-caching-docker-layers-dependency-cache



Webstack Builders
1032 E. Brandon Boulevard, Suite 5230
Brandon, FL 33511

support@webstackbuilders.com
Toll Free (888) 987-1881
Local (302) 608-6864