

Build Systems

Aayush Bajaj

July 10, 2026

Contents

1 the model: a dag of targets	1
2 make: the actual semantics	2
3 ninja: the assembler	3
4 incremental correctness	4
4.1 mtimes and their failure modes	4
4.2 content hashing	4
5 hermeticity: the bazel model	5
6 caching	5
7 the dag inside the language	5
8 see also	6

every build system — make, ninja, bazel, cargo, even npm scripts pretending otherwise — is the same machine: a directed acyclic graph of files and commands, plus a policy for deciding which part of the graph is stale. everything else is syntax.

1 the model: a dag of targets

- a **target** is a file the build can produce; a **rule** says how (a command) and from what (its **dependencies**).
- dependencies point from outputs to inputs; because outputs of one rule are inputs to another, the whole thing is a dag — a cycle would mean “to build a you must first build a”.
- a build is then two steps:
 1. **mark dirty**: a target is dirty if it is missing, if any dependency is dirty, or if the staleness policy (timestamps or hashes, below) says an input changed. dirtiness propagates along edges — one flipped source poisons its whole downstream cone.
 2. **evaluate in topological order**: run each dirty target’s command after all its dependencies are up to date (Cormen, Thomas H. and Leiserson, Charles E. and Rivest, Ronald L. and Stein, Clifford, 2009). independent dirty targets can run in parallel (**-j8**) — the dag *is* the parallelism plan,

which is why builds are embarrassingly parallel until the link step serialises everything.

- the entire correctness contract: **the graph must be complete**. every undeclared dependency is a future “works after `make clean`” bug — the build system faithfully skips rebuilding things it was never told could change.

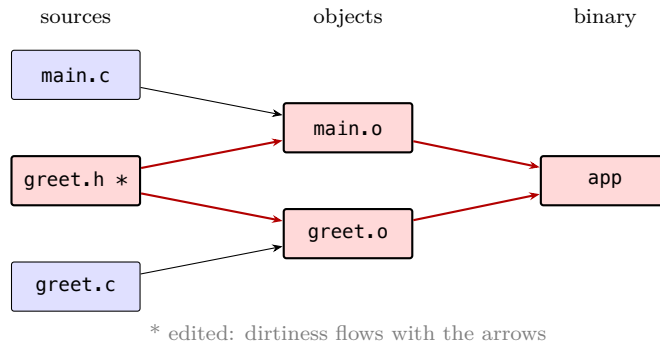


Figure 1: dirty-node propagation: editing `greet.h` dirties every target downstream of it (red); `main.c` and its untouched cone stay cached. the build re-runs exactly the red commands, in topological order.

2 make: the actual semantics

`make`’s whole rebuild rule fits in one sentence: **a target is rebuilt if it does not exist, or if any prerequisite’s mtime is newer than the target’s mtime**. everything else is macro expansion.

- a rule is **target: prerequisites** + tab-indented recipe lines (the tab is load-bearing; four spaces is a syntax error that has wasted a cumulative civilisation of hours).
- automatic variables inside recipes: `$@` the target, `$<` the first prerequisite, `$$` all prerequisites.
- pattern rules** (`%.o: %.c`) define whole families of edges at once; `make` instantiates them by stem-matching.
- phony targets** (`clean`, `all`, `test`) are commands masquerading as files. declare them `.PHONY:` or a stray file literally named `clean` in the working directory will silently satisfy the target and nothing will run.

a real one, executed:

```

1 CC      := cc
2 CFLAGS  := -Wall -O2
3
4 app: main.o greet.o           # link rule: target depends on objects
5     $(CC) $$^ -o $@
6
7 %.o: %.c greet.h             # pattern rule: any .o from its .c (+ the header)
8     $(CC) $(CFLAGS) -c $< -o $@
9
10 .PHONY: clean
11 clean:
12     rm -f app *.o

```

```

== first build:
cc -Wall -O2 -c main.c -o main.o
cc -Wall -O2 -c greet.c -o greet.o
cc main.o greet.o -o app
== second build (nothing changed):
make: `app' is up to date.
== touch greet.c, rebuild:
cc -Wall -O2 -c greet.c -o greet.o
cc main.o greet.o -o app
== run it:
hello from make

```

exactly the dag story: touching `greet.c` dirtied `greet.o` and therefore `app`, and `main.o` was never recompiled.

- the classic hole: header dependencies. hardcoding `greet.h` into the pattern rule (as above) over-rebuilds; omitting it under-rebuilds. real makefiles have the compiler emit the true per-file dependency list (`-MMD -MP` producing `.d` fragments, then `-include *.d`) — the graph is extracted from the source rather than maintained by hand.
- second classic hole: **recursive make** (`$(MAKE) -C subdir`) fragments the dag into islands; no single process sees the whole graph, so parallelism stalls at directory boundaries and cross-directory dependencies go undeclared.

the 1997 miller paper ‘recursive make considered harmful’ — the rare ‘considered harmful’ title that was simply correct

3 ninja: the assembler

ninja’s insight is a division of labour: humans should not write build files, *programs* should — so the build-time language can be stripped to almost nothing and evaluated fast.

- no globbing, no pattern matching, no conditionals, no functions at build time: `build.ninja` lists every edge explicitly. all cleverness happens once, in the **generator** (cmake, meson, gn — or a 200-line script), whose output is dumb and therefore instantly loadable. on no-op builds ninja starts, stats the graph, and exits in milliseconds — which is the operation developers run hundreds of times a day.
- this site eats its own dog food: `scripts/build.py` globs the org sources, emits a `build.ninja` with one `org2md` edge per page (`build content/x.md: org2md content-org/x.org`), and hands it to ninja — regenerate-then-build, the generator/assembler split in two hundred lines of python.
- the same toy graph, hand-written this time and executed:

```

1 cflags = -Wall -O2
2
3 rule cc
4     command = cc $cflags -c $in -o $out
5     description = CC $out
6
7 rule link
8     command = cc $in -o $out
9     description = LINK $out
10
11 build main.o: cc main.c
12 build greet.o: cc greet.c

```

```
13 | build app: link main.o greet.o
```

```
== first build:
[1/3] CC greet.o
[2/3] CC main.o
[3/3] LINK app
== second build:
ninja: no work to do.
== touch main.c:
[1/2] CC main.o
[2/2] LINK app
```

- niceties earned by the strict model: correct parallelism by default (it saturates cores without `-j` guessing), a build log that records the command line per edge (change the flags, rebuild — make happily won't), `ninja -t graph` to dump the dag as graphviz, and header deps ingested from the compiler (`deps = gcc`) into a compact binary cache.

4 incremental correctness

the staleness oracle is where build systems quietly lie to you.

4.1 mtimes and their failure modes

- comparing timestamps is $O(1)$ per edge and needs no state — and is wrong in several boring, real ways:
 - **granularity**: filesystems and tools that store whole-second mtimes make edit-within-the-same-second invisible. this page's make demo genuinely hit it — the first run compiled, touched and re-ran `make` inside one second, and make 3.81 reported `app is up to date`; the transcript above needed a `sleep 1` before the `touch`.
 - **clock skew**: an nfs server or ci cache whose clock runs ahead writes files “from the future”; every subsequent local edit looks older than the stale output and nothing ever rebuilds. distributed timestamps are not comparable — the same lesson [distributed systems](#) keep re-teaching.
 - **content-free rebuilds**: `git checkout`, `touch`, or re-saving an identical file bumps mtime and triggers a full downstream rebuild for zero semantic change.
- mtime also ignores *how* the output was produced: change `CFLAGS` and make sees nothing stale. (ninja's command-line log closes exactly this hole.)

4.2 content hashing

- hash the inputs (and the command line); rebuild when the hash changes. immune to granularity, skew and touch-storms; identical content never rebuilds.
- cost: reading and hashing every input on every build — mitigated by caching hashes keyed on (mtime, size, inode) so the expensive path only runs when the cheap heuristic fires. bazel and ccache both live here.
- the hash also unlocks **caching** (next section): if the action's inputs hash to something seen before, the output can be fetched instead of built.

5 hermeticity: the bazel model

one honest section, because the idea matters beyond bazel:

- an action is **hermetic** if its output depends only on declared inputs — not on the machine’s environment, installed toolchains, the network, or the clock. bazel enforces this by sandboxing every action (it literally cannot read undeclared files) and pinning toolchains as inputs.
- hermetic + deterministic actions give you, essentially for free: **remote caching** (a content-addressed action cache shared by the whole team — your colleague already built this object), **remote execution** (fan the dag out over a farm), and byte-for-byte **reproducible builds** (the security property: anyone can verify the shipped binary matches the source).
- the honest price: you must declare *everything*, vendored or pinned; the ecosystem-native package managers fight you; **BUILD** file maintenance is a tax (partly automated); and the first-build experience is slower and stranger than **make**. the model pays for itself roughly when repo size times team size makes “clean build in ci every time” intolerable — well before that, ninja-plus-a-generator is the sweet spot.

6 caching

- **ccache**: wraps the compiler, hashes the preprocessed source + flags, and serves the object file from a local cache on a hit. rescues the “clean build” habit and branch-switching without any build-system cooperation.
- **remote/action caches** (bazel, buck2, gradle, turborepo lineage): key = hash of (command, input tree); value = output artefacts. correctness inherits everything from hermeticity — a cache in front of non-hermetic actions is a machine for distributing other people’s wrong outputs at scale.
- cache *invalidation* is not the hard part here — the key contains everything by construction. the hard part is key completeness, which is hermeticity again.

7 the dag inside the language

the graph does not stop at file boundaries; the compiler has its own.

- c/c++: each **.c** file is a **compilation unit**, compiled independently against textually **#include**-d headers (Kernighan, Brian W. and Ritchie, Dennis M., 1988). headers are the fan-out disaster: a popular header included by 2000 units means a one-line edit recompiles 2000 files. hygiene — forward declarations, include-what-you-use, pimpl — is dependency-graph surgery, the same **coupling** argument at compile-time scale. c++20 modules replace textual inclusion with real compiled interfaces.
- **lto** (link-time optimisation) bends the model: objects carry compiler ir instead of final code, and cross-unit inlining happens at link time — better code, but the link step inherits a large serial chunk of the work the dag used to parallelise (thin-lto claws most of it back).
- interpreted ecosystems have the same graph wearing a different hat: python imports, javascript bundlers tree-shaking module graphs, and lockfiles pinning

the dependency dag of packages rather than files.

8 see also

- [concurrency](#) — `-j8` is a scheduling problem; the dag is the lock-free plan
- [containers](#) — image layer caching is mtime-vs-hash staleness, third costume
- [data structures & algorithms](#) — topological sort, the engine under all of it
- [testing](#) — incremental test selection is the same dirty-cone computation
- [design patterns](#) — the generator/assembler split is builder thinking applied to builds

references

Cormen, Thomas H. and Leiserson, Charles E. and Rivest, Ronald L. and Stein, Clifford (2009). *Introduction to Algorithms*, MIT Press.

Kernighan, Brian W. and Ritchie, Dennis M. (1988). *The C Programming Language*, Prentice Hall.