Engineer

Semantic versioning

a version-number scheme that signals breaking changes

What it is

Semantic versioning, usually shortened to SemVer, is a set of rules for numbering releases so that the number itself says what kind of change a release contains. The specification, now at version 2.0.0 and originally written by Tom Preston-Werner, uses three numbers in the form MAJOR.MINOR.PATCH. A release that breaks the public API raises the major number, one that adds functionality in a backward-compatible way raises the minor number, and one that only fixes bugs without touching the API raises the patch number. When a number goes up, the numbers to its right go back to zero, and each is compared as a number rather than as text, so `1.10.0` comes after `1.9.0` .

Everything rests on one requirement: the project must declare a public API, in the code or in its documentation, because the numbers describe changes to that API and nothing else. Once a version is released, its contents must not change; a correction is a new version. Major version zero, `0.y.z` , is for initial development, where anything may change at any time and the API should not be considered stable. Version `1.0.0` is the point at which the public API is defined, and from then on the numbers follow how it changes.

Two optional labels extend the format. A pre-release goes after a hyphen and sorts before the release it leads up to, so `1.0.0-alpha` comes before `1.0.0-alpha.1` , which comes before `1.0.0-beta` , `1.0.0-rc.1` and finally `1.0.0` . Build metadata goes after a plus sign, as in `1.0.0+20130313144700` , and is ignored when versions are put in order, so two versions that differ only there rank the same. A leading v, as in the Git tag `v1.2.3` , is not part of the semantic version; the specification calls it a common way of marking a version number. Go makes it mandatory: a canonical Go module version is the letter v followed by a semantic version.

Use the concept in a decision

Put “What is semantic versioning?” to work in a real decision. We can help relate the concept to the system you are building or buying. Tested, released and reversible, so changes reach customers on a Tuesday afternoon rather than late on a Friday night.

Explore our services Get in touch →

Test the assumptions behind the term

Use the explanation and linked work to frame questions for your project. We can examine the assumptions with you before a tool is chosen. Automated GIS SaaS deployment with GitHub Actions — deployment, data and reporting, all off the hands

Automated GIS SaaS deployment with GitHub Actions Get in touch →

Why it matters

The specification starts from what it calls dependency hell. Specify dependencies too tightly and you get version lock, where upgrading one package means releasing new versions of everything that depends on it. Specify them too loosely and you get what the specification calls version promiscuity: accepting future releases without a sound basis for expecting compatibility. A version number with an agreed meaning lets a project state what it will accept: a library that needs a function first added in 3.1.0 can accept anything from 3.1.0 up to, but not including, 4.0.0, and take later fixes and additions without anyone editing the requirement.

Package managers act on that promise without asking. In Cargo, the Rust package manager, a plain requirement such as `"1.2.3"` is a caret requirement and means at least 1.2.3 but below 2.0.0. When it writes `Cargo.lock`, Cargo picks the greatest version inside that range, and it stays on the 1.x line after 2.0.0 is published, because it treats 2.0.0 as incompatible. In node-semver, which calls itself the semantic versioner for npm, `^1.2.3` means `>=1.2.3 <2.0.0-0` and `~1.2.3` allows only patch-level changes, `>=1.2.3 <1.3.0-0`. Go builds the rule into import paths: from major version 2 onward a module’s path must end in a suffix such as `/v2`, so two incompatible major versions are two different import paths. Go’s minimal version selection then takes, for each module, the highest version that any requirement asks for, which is the minimum that satisfies all of them. The result is not saved in a lock file; every module-aware command computes it again.

None of this makes the number a guarantee. It is a promise made by the people who publish the package, and it is only as good as their judgement about what counts as breaking, which is why Cargo publishes a long compatibility guide that sorts changes into major, minor and possibly-breaking. Most of that guide is about changes that stop code that compiled yesterday from compiling today.

The guide itself notes that almost every change carries some risk to runtime behaviour, and that whether such a change is incompatible is usually a judgement call for the maintainers.

What it looks like in practice

Follow an invented Rust library, tally, through its releases. Version 0.4.2 is still initial development. Version 1.0.0 declares the public API stable. Version 1.0.1 fixes a rounding bug and is a patch. Version 1.1.0 adds a function, `parse_strict`, and marks the old `parse` as deprecated: the specification requires a minor release for a deprecation and asks for at least one such release before the function is removed. Version 2.0.0 removes `parse` and is a major release. Version 2.1.0-rc.1 is a pre-release of the next minor version.

An application that depends on tally and three other invented crates states its requirements in `Cargo.toml`. The comments give the range Cargo reads from each line:

```
[dependencies]
tally = "1.0.1"         # >=1.0.1, <2.0.0: takes 1.1.0, never 2.0.0
ledger = "0.4.2"        # >=0.4.2, <0.5.0: 0.5.0 counts as breaking
stamp = "~1.2.3"        # >=1.2.3, <1.3.0: patch releases only
quill = "=2.1.0-rc.1"   # exactly this pre-release
```

The `ledger` line shows a convention of the tools rather than of the specification: Cargo treats the left-most non-zero number as the one that signals a break, where the specification says everything below 1.0.0 may change at any time. Cargo leaves pre-releases out of a requirement unless it names one, so a requirement of `"2.0"` would never pick up tally's release candidate; the exact `=` form on the last line is the case where Cargo's documentation says a fully specified requirement might be warranted: depending on a pre-release while avoiding breaking changes between pre-releases. Once resolved, the versions are recorded in `Cargo.lock`, and `cargo update` moves them within these ranges. With `--locked`, a build that would have to change the lock file returns an error instead.

What the number does not cover is anything outside the API. The licence audit of 644 Rust crates, carried out in September 2026, checks the licence of every crate on every build against an allow list, because a routine version bump deep in the dependency graph can change what an application may be distributed under,

silently, in a lock file that nobody reads in full. To prove the check works, it was made to fail twice on purpose. One test rewrote a crate's licence expression, the shape of a crate changing its terms upstream between versions, which human review misses because the crate itself is not new. The other moved the pinned core revision without regenerating the audit. A semantic version can tell you that 1.4.0 is meant to work with code written against 1.0.0. It cannot tell you that you may still ship it.

Where it goes wrong

A breaking change in a minor release. It happens by accident or through a different view of what counts as breaking. Cargo's guide lists changes that look harmless and are not: adding a variant to a public enum breaks code that matches on it exhaustively, unless the enum was marked `#[non_exhaustive]` from the start, and in rare cases involving glob imports even adding a new public item causes a compile error. The specification's answer, once the mistake is noticed, is to release a new minor version that restores compatibility, never to modify the release already published, and to tell users which version was at fault. On the consuming side, a committed lock file and tests that run on every update are what catch it before production does.

Staying at 0.x for years. Under the specification, major version zero carries no stable API at all. Its FAQ is blunt: "If your software is being used in production, it should probably already be 1.0.0." The tools fill the gap with their own convention: Cargo and node-semver treat the left-most non-zero number as the breaking one, so `^0.2.3` accepts only further 0.2.x releases and `^0.0.3` accepts no updates at all. A library that has sat at 0.x for years has users who depend on it but no defined promise to them. Release 1.0.0 once people rely on the API; it changes a number, not the code.

Reading more into the number than it says. Compatibility under the specification is about the public API. Cargo's guide classes raising the minimum Rust version a crate needs as possibly-breaking and generally recommends treating it as a minor change, so a minor update can stop the build on an older toolchain. Setting `package.rust-version` lets Cargo's resolver try older versions of the crate when needed. Licence terms, performance and behaviour the API never documented are outside the number altogether. Check them separately, with a licence audit and tests, rather than inferring them from a minor release.

Two major versions in one build. When two dependencies need incompatible versions of the same crate, say 0.6 and 0.7, Cargo builds both. The compiler treats their types as different even when the names match. That usually shows up as a compile error and sometimes only at runtime: Cargo's documentation describes a downcast that fails because the object was created by a library using version 1.0.0 while the calling code downcasts to the type from 2.0.0. `cargo tree -d` lists duplicate versions and where they come from. A library author can soften a major release with what Cargo's documentation calls the semver trick: publish the new major version, then a point release of the old one that re-exports the types from the new.

Further reading

- Semantic Versioning 2.0.0 — the specification and its FAQ, including what to do after releasing a breaking change by mistake.
- SemVer Compatibility — the Cargo Book's guide to which changes in a Rust library are major, minor or possibly-breaking, each with an example.
- Specifying Dependencies — Cargo's requirement syntax: default, caret, tilde and comparison requirements, and how pre-releases are matched.
- Dependency Resolution — how Cargo picks versions, what the lock file holds and the hazards of two incompatible versions in one graph.
- node-semver — the semantic versioner for npm, with its caret, tilde and pre-release range rules.
- Go Modules Reference — major version suffixes, the import compatibility rule and minimal version selection.

Glossary

Choose for your actual constraints

Bring the deployment path, the operational risks and the people on call. We can scope a change with checks, a recovery route and clear ownership.

See what it costs Get in touch →

Make the terminology practical

Ask how “What is semantic versioning?” applies to your organisation. A concrete example from your current system is enough to start the discussion.

Explore our services

welcome@engineer.company +45 50 10 61 84 Copenhagen, Denmark

Take this page with you

Contact card

Carefully Crafted With Love by D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

Copyright © 2025 – present · Engineer ApS · All rights reserved.

Printed from https://engineer.company/glossary/semantic-versioning/