Engineer

Go

a compiled, strongly typed, garbage-collected language

What it is

Go is a general-purpose programming language that began at Google, designed with systems programming in mind and with concurrency written into the language itself rather than added as a library. It is often called golang, after the project's first web address, golang.org; the project's FAQ is firm that the language is just Go, written Go and never GO.

Go's goals were first sketched on a whiteboard by Robert Griesemer, Rob Pike and Ken Thompson on September 21, 2007, and on November 10, 2009, the language was released to the public as an open-source project. Go 1 followed on March 28, 2012, with a promise that programs written for it would keep compiling and running, unchanged, under every later Go 1 release. That promise still holds: the specification now describes language version go1.27, and Go 1.27.0 was released on August 19, 2026. Generics arrived in Go 1.18, and the language's designers called them the biggest change since the first open-source release.

The FAQ says the language grew out of a choice its authors were tired of making, between efficient compilation, efficient execution and ease of programming. Go sets out to give all three. A program is built from packages, and the toolchain is one command, `go`, which builds, tests, formats and fetches dependencies. The keyword `go` in front of a function call starts a goroutine, a function running alongside the rest of the program, and goroutines are cheap enough that, the FAQ says, hundreds of thousands can share one address space. Channels carry values between them, and Effective Go sums up the intended style in a slogan: "Do not communicate by sharing memory; instead, share memory by communicating."

Two habits mark Go code wherever it is found. The `gofmt` program prints every source file in one standard layout, so formatting is never argued about in review. And the compiler refuses a program with an unused variable or an unused import, because the first may hide a bug and the second slows every later build.

Use the concept in a decision

Put “What is Go?” to work in a real decision. We can help relate the concept to the system you are building or buying. The part users never see: where the data lives, what the rules are, and how other systems are allowed to talk to it.

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. Designed and ran 6 ETL/ELT pipelines — 6 ETL pipelines across three database engines

Designed and ran 6 ETL/ELT pipelines Get in touch →

Why it matters

The decision Go answers is usually what to write a network service, a command-line tool or a piece of infrastructure in, and the alternatives sit on either side of it: a scripting language on one, C, C++ or Rust on the other.

Against a scripting language, Go offers simple deployment and early errors. The linker produces statically linked binaries by default, so the output of a build is one file that carries its own runtime and runs on a server with nothing else installed. The FAQ puts the price at a couple of megabytes for a program that prints “hello, world”, against about 750 kB for the same program in C. Type errors surface when the program is compiled rather than when a rarely used branch runs in production, and one founding goal was that even a large executable should build in a few seconds on a single computer.

Against C, C++ or Rust, Go offers simplicity and memory safety at the cost of control. A garbage collector reclaims memory, so ordinary code is spared a whole class of use-after-free mistakes, but the program does not decide exactly when memory is released, and every binary ships the runtime that does it. For most servers that is the right trade. For a kernel, a small embedded device or a hot loop that cannot tolerate a collector, it is the wrong one.

The quieter argument is time. In 2023 Russ Cox wrote that a Go 2 that breaks old programs “is never going to happen”, and that the project would “double down on compatibility” instead. The promise is paid for in upgrades rather than

rewrites. A new major release ships every six months, in February and August, and each is supported until two newer ones exist, so a team that wants security fixes moves forward about once a year.

The small language also writes some things out longhand. Errors are ordinary values, returned and checked where they occur, which is explicit and repetitive in about equal measure.

What it looks like in practice

A goroutine is cheap to start and just as cheap to lose. The page on moving slow work onto a River job queue says so plainly: do slow work in a goroutine and it disappears the moment the process restarts, mid-deploy, with no record that it was ever supposed to happen. The company moved that work onto River, which keeps its queue in PostgreSQL, so a job and the rows it touches commit or roll back together. Behind it sit 15 worker modules and 8 scheduled tasks, with 20 pg_cron jobs underneath. A goroutine gives concurrency; durability has to come from somewhere else.

Testing ships with the toolchain, and the layered test suite shows what that adds up to. Its Go API carries 981 test functions across 310 files, one of four layers that are each tested where their behaviour lives, beside the frontend's specs and the database's own SQL tests.

The continuous security programme leans on the language's tooling too. Go dependencies are checked with govulncheck, and a vulnerability recognised by the configured checks can fail the build. The page names the volume of findings as the real cost of scanning, and govulncheck is designed against that cost: it reports a vulnerability only when the code's own functions call the affected function, directly or through others.

Where it goes wrong

A function reports an error when nothing went wrong. An interface value in Go holds a type and a value, and it equals nil only when both are unset. A function that returns a nil pointer of a concrete error type through the `error` interface hands its caller an interface that holds a type, and that interface is not nil:

```
func save() error {
    var err *ValidationError // a nil pointer
    if invalid() {
        err = &ValidationError{}
    }
    return err // never equal to nil: the interface holds a type
}
```

Return a literal `nil` on success, and declare every error result as `error` rather than as a concrete type.

A service that passed every test crashes under real traffic. Map operations are deliberately left unsynchronised, because locking every access would slow most programs, and the FAQ warns that “uncontrolled map access can crash the program”. The cause is one goroutine writing to a map while another reads or writes it; reads alone are safe. Guard the map with a mutex, give it to a single goroutine that the others reach through a channel, or use `sync.Map` for the cases it was designed for. Then run the tests with `go test -race` . For a typical program the race detector can raise memory use five to ten times and run time two to twenty times, and it finds only races on code paths that actually run, so a binary built with `-race` and put under a realistic load finds more.

Goroutines started in a loop all work on the last item. Before Go 1.22 a `for` loop had one variable shared by every iteration, so a closure that captured it saw whatever value it held when the goroutine finally ran. Go 1.22 gives each iteration its own variable, but only in modules whose go.mod declares `go 1.22` or later, and older code keeps its old meaning even when a newer toolchain builds it. Raise the `go` line and run `go vet` and the tests. A test that starts failing after the change was often checking the wrong value all along.

A build on a fresh machine stops to download a different Go. Since Go 1.21 the `go` line is a requirement, and when the installed toolchain is older than a module asks for, the `go` command fetches the newer one as a module and runs it from the module cache. On a laptop that is convenient; in an air-gapped build or a pinned container image it is a surprise. Install the toolchain the go.mod names, and set the `GOTOOLCHAIN` environment variable when a build has to use one exact version.

Further reading

- The Go Programming Language Specification — the reference manual for every construct in the language, versioned with each release.
- Frequently Asked Questions (FAQ) — why the language was made, the design decisions behind it, and the nil-error and map puzzles above.
- Go 1 and the Future of Go Programs — the compatibility promise and the exceptions it allows.
- Release History — every release since Go 1, with the support policy at the top.
- Data Race Detector — how to run it, what it catches and what it costs.
- Fixing For Loops in Go 1.22 — the loop-variable change, why it is tied to go.mod, and how to test old code against it.

Glossary

Choose for your actual constraints

Describe the workflow, the existing interfaces and the users who need the change. We can define a testable delivery and a handover that lets your team maintain it.

See what it costs Get in touch →

Make the terminology practical

Ask how “What is Go?” 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/go/