Released as open source in November 2009, Go has become the foundation for critical infrastructure at every major cloud provider. Its creators look back on how Go got here and why it has stuck around.

BY RUSS COX, ROBERT GRIESEMER, ROB PIKE, IAN LANCE TAYLOR, AND KEN THOMPSON

The Go Programming Language and Environment

GO IS A programming language created at Google in late 2007 and released as open source in November 2009. Since then, it has operated as a public project, with contributions from thousands of individuals and dozens of companies. Go has become a popular language for building cloud infrastructure: Docker, a Linux container manager, and Kubernetes, a container deployment system, are core cloud technologies written in Go. Today, Go is the foundation for critical infrastructure at every major cloud provider and is the implementation language for most projects hosted at the Cloud Native Computing Foundation.

Early users were attracted to Go for a variety of reasons. A garbage-collected, statically compiled language for building systems was unusual. Go's native support for concurrency and parallelism helped take advantage of the multicore machines that were becoming mainstream at the time. Self-contained binaries and easy cross-compilation simplified deployment. And Google's name was undoubtedly a draw.

But why did users stay? Why has Go grown in popularity when so many other language projects have not? We believe that the language itself forms only a small part of the answer. The full story must involve the entire Go environment: the libraries, tools, conventions, and overall approach to software engineering, which all support programming in the language. The most important decisions made in the language's design, then, were the ones that made Go better for large-scale software engineering and helped us attract like-minded developers.

In this article, we examine the design decisions we believe are most responsible for Go's success, exploring how they apply not just to the language but also to the environment more broadly. It is difficult to isolate the contributions of any specific decision, so this article should be read not as scientific analysis, but as a presentation of our best understanding, based on

» key insights

- The Go language enjoys widespread adoption despite having few technical advances. Instead, Go succeeded by focusing on the overall environment for engineering software projects.

- Go's approach is to treat language features as no more important than environmental ones, such as careful handling of dependencies, scalable development and production, programs that are secure by default, tool-aided testing and development, amenability to automated changes, and long-term guaranteed compatibility.

- Go 1.18, released in March 2022, added its first major new language feature in a decade: parametric polymorphism tailored to fit well with the rest of Go.

experience and user feedback over the past decade of Go.

Origins

Go arose through experience building large-scale distributed systems at Google, working in a large codebase shared by thousands of software engineers. We hoped that a language and tools designed for such an environment could address challenges faced by the company and industry at large. Challenges arose due to the scale of both the development efforts and the production systems being deployed.

Development scale. On the development side, Google in 2007 had about 4,000 active users working in a single, shared, multi-language (C++, Java, Python) codebase.[3] The single codebase made it easy to fix, for example, a problem in the memory allocator that was slowing down the main web server. But when working on a library, it was too easy to unwittingly break a previously unknown client because of the difficulty of finding all the dependencies of a package.

Also, in existing languages we used, importing one library could cause the compiler to recursively load all the libraries that one imported. In one C++ compilation in 2007, we observed the compiler (after `#include` processing) reading more than 8 GB of data when handed a set of files totaling 4.2 MB, an expansion factor of almost 2,000 on an already large program. If the number of header files read to compile a given source file grows linearly with the source tree, the compilation cost for the entire tree grows quadratically.

To compensate for the slowdown, work began on a new, massively parallel and cacheable build system, which eventually became the open source Bazel build system.[23] But parallelism and caching can do only so much to repair an inefficient system. We believed the language itself needed to do more to help.

Production scale. On the production side, Google was running very large systems. For example, in March 2005, one 1,500-CPU cluster of the Sawzall log analysis system processed 2.8 PB of data.[26] In August 2006, Google's 388 Bigtable serving clusters comprised 24,500 individual tablet servers, with one group of 8,069 servers handling an aggregate 1.2 million requests per second.[4]

Yet Google, along with the rest of the industry, was struggling to write efficient programs to take full advantage of multicore systems. Many of our systems resorted to running multiple copies of the same binary on a single machine, because existing multithreading support was both cumbersome and low performance. Large, fixed-size thread stacks, heavyweight stack switches, and awkward syntax for creating new threads and managing interactions between them all made it more difficult to use multicore systems. But it was clear that the number of cores in a server was only going to grow.

Here too, we believed that the language itself could help, by providing lightweight, easy-to-use primitives for concurrency. We also saw an opportunity in those additional cores: a garbage collector could run in parallel with the main program on a dedicated core, reducing its latency costs.

Go is our answer to the question of what a language designed to meet these challenges might look like. Part of Go's popularity is undoubtedly that the entire tech industry now faces these challenges daily. Cloud providers make it possible for even the smallest companies to target very large production deployments. And while most companies do not have thousands of active employees writing code, almost all companies now depend on large amounts of open source infrastructure worked on by thousands of programmers.

The remainder of this article examines how specific design decisions address these goals of scaling both development and production. We start with the core language itself and work outward to the surrounding environment. We do not attempt to give a complete introduction to the language. For that, see the Go language specification[18] or books such as *The Go Programming Language*.[11]

Packages

A Go program is made up of one or more importable packages, each containing one or more files. The web server in Figure 1 illustrates many important details about the design of Go's package system:

The program starts a local web server (line 9) that handles each request by calling the hello function, which responds with the message "hello, world" (line 14).

A package imports another using an explicit import statement (lines 3-6), as in many languages but in contrast to C++'s textual #include mechanism. Unlike most languages, though, Go arranges that each import reads only a single file. For example, the fmt package's public API references types from the io package: the first argument to fmt.Fprintf is an interface value of type io.Writer. In most languages, a compiler processing the import of fmt would also load all of io to make sense of fmt's definitions, which might in turn require loading additional packages to make sense of all of io's definitions. A single import statement could end up processing tens or hundreds of packages.

Go avoids this work by arranging, similar to Modula-2,[13] for the compiled fmt package's metadata to contain everything necessary to know about its own dependencies, such as the definition of io.Writer. Thus, the compilation of import "fmt" reads only a single file that completely describes fmt and its dependencies. Moreover, this flattening is done once, when fmt is compiled, avoiding many loads each time it is imported. This approach leads to less work for the compiler and faster builds, helping large-scale development. Also, package import cycles are disallowed: since fmt imports io, io cannot import fmt, nor anything else that imports fmt, even indirectly. This too leads to less work for the compiler, guaranteeing that a particular build can be split up at the level of individual, separately compiled packages. This also enables incremental program analyses, which we run to catch mistakes even before running tests, as described below.

Importing fmt does not make the name io.Writer available to the client. If the main package wants to use the type io.Writer, it must import "io" for itself. Thus, once all references to fmt-qualified names have been removed from the source file—for example, if the fmt.Fprintf call is deleted—the import "fmt" statement is safe to remove from the source without further analysis. This property makes it possible to automate management

Figure 1. A Go web server.

```
1  package main
2
3  import (
4      "fmt"
5      "net/http"
6  )
7
8  func main() {
9      http.ListenAndServe("localhost:8080",
10         http.HandlerFunc(hello))
11 }
12
13 func hello(w http.ResponseWriter, req *http.Request) {
14     fmt.Fprintf(w, "hello, world\n")
15 }
```

Figure 2. The io package's writer interface.

```
type Writer interface {
    Write(data []byte) (count int, err error)
}
```

of imports in the source code. In fact, Go disallows unused imports to avoid bloat created by linking unused code into programs.

Import paths are quoted string literals, which enables flexibility in their interpretation. A slash-separated path identifies the imported package in the `import`, but then source code refers to the package using the short identifier declared in the package statement. For example, `import "net/http"` declares the top-level name `http` that provides access to its contents. Beyond the standard library, packages are identified by URL-like paths beginning with domain names, as in `import "github.com/google/uuid"`. We will have more to say about such packages later.

As a final detail, note the uppercase letter in the names `fmt.Fprintf` and `io.Writer`. Go's analog of C++ and Java's public, private, and protected concepts and keywords is a naming convention. Names with a leading uppercase letter, such as `Printf` and `Writer`, are "exported" (public). Others are not. The case-based, compiler-enforced export rule applies to package-level identifiers for constants, functions, and types; method names; and struct field names. We settled on this rule to avoid the syntactic weight of having to write a keyword like `export` next to every identifier involved in the public API. Over time, we have come to value the ability to see whether the identifier is available outside the package or is purely internal at each of its uses.

Types

Go provides the usual set of basic types: Booleans, sized integers such as `uint8` and `int32`, unsized `int` and `uint` (32- or 64-bit, depending on machine size), and sized floating-point and complex numbers. It provides pointers, fixed-size arrays, and structs in a manner similar to C. It also provides a built-in string type, a hash table called a map, and dynamically sized arrays called slices. Most Go programs rely on these and no other special container types.

Go does not define classes but allows methods to be bound to any type, including structs, arrays, slices, maps, and even basic types, such as integers. It does not have a type hierarchy; we

Today, Go is the foundation for critical infrastructure at every major cloud provider.

felt that inheritance tended to make programs harder to adapt as they grow. Instead, Go encourages composition of types.[9]

Go provides object-oriented polymorphism through its interface types. Like a Java interface or a C++ abstract virtual class, a Go interface contains a list of method names and signatures. For example, the `io.Writer` interface mentioned earlier is defined in the `io` package as shown in Figure 2.

`Write` accepts a slice of bytes and returns an integer and possible error. Unlike in Java and C++, any Go type that has methods with the same names and signatures as an interface is considered to implement that interface, without explicitly declaring that it does so. For example, the type `os.File` has a `Write` method with the same signature, and therefore it implements `io.Writer`, without an explicit signal like Java's "implements" annotations.

Avoiding the explicit association between interfaces and implementations allows Go programmers to define small, nimble, often ad hoc interfaces, rather than using them as foundation blocks in a complex type hierarchy. It encourages capturing relationships and operations as they arise during development, instead of needing to plan and define them all ahead of time. This especially helps with large programs, in which the eventual structure is much more difficult to see clearly when first beginning development. Removing the bookkeeping of declaring implementations encourages the use of precise, one- or two-method interfaces, such as `Writer`, `Reader`, `Stringer` (analogous to Java's `toString` method), and so on, which pervade the standard library.

Developers first learning about Go often worry about a type accidentally implementing an interface. Although it is easy to build hypotheticals, in practice it is unlikely that the same name and signature would be chosen for two incompatible operations, and we have never seen it happen in real Go programs.

Concurrency

When we started designing Go, multicore computers were becoming widely

available, but threads remained a heavyweight concept in all popular languages and operating systems. The difficulty of creating, using, and managing threads made them unpopular,[24] limiting access to the full power of multicore CPUs. Resolving this tension was one of the prime motivations for creating Go.

Go includes in the language itself the concept of multiple concurrent threads of control, called *goroutines*, running in a single shared address space and efficiently multiplexed onto operating system threads. A call to a blocking operation, such as reading from a file or network, blocks only the goroutine doing the operation; other goroutines on the thread may be moved to another thread so they can continue to execute while the caller is blocked. Goroutines start with only a few kilobytes of stack, which is resized as needed, without programmer involvement. Developers use goroutines as a plentiful, inexpensive primitive for structuring programs. It is routine for a server program to have thousands or even millions of goroutines, as they are much cheaper than threads.

For example, `net.Listener` is an interface with an `Accept` method that can listen for and return new incoming network connections. Figure 3 shows a function `listen` that accepts connections and starts a new goroutine to run the serve function for each.

The infinite `for` loop in the listen function body (lines 22–28) calls `listener.Accept`, which returns two values: the connection and a possible error. Assuming there is no error, the `go` statement (line 27) starts its argument—the function call `serve(conn)`—in a new goroutine, analogous to the & suffix to a Unix shell command but inside the same operating system process. The function to be called as well as its arguments are evaluated in the original goroutine; those values are copied to create the initial stack frame of the new goroutine. Thus, the program runs an independent instance of the `serve` function for each incoming network connection. An invocation of `serve` handles the requests on a given connection one at a time (the call to `handle(req)` on line 37 is not prefixed by `go`); each call can block with-

out affecting the handling of other network connections.

Under the hood, the Go implementation uses an efficient multiplexing operation, such as Linux's epoll, to handle concurrent I/O operations, but the user doesn't see that. The Go runtime library instead presents the abstraction of blocking I/O, in which each goroutine executes sequentially—no callbacks needed—which is easy to reason about.

Having created multiple goroutines, a program must often coordinate between them. Go provides *channels*, which allow communication and synchronization between goroutines: a channel is a unidirectional, limited-size pipe carrying typed messages between goroutines. Go also provides a multiway `select` primitive that can control execution according to which communications can proceed. These ideas are adapted from Hoare's "Communicating Sequential Processes"[19] and earlier language experiments, specifically Newsqueak,[25] Alef,[31] and Limbo.[12]

Figure 4 shows an alternate version of `listen`, written to limit the number of connections served at any moment.

This version of `listen` begins by creating a channel named `ch` (line 42) and then starting a pool of 10 server goroutines (lines 44–46), which receive connections from that single channel. As new connections are accepted, `listen` sends each on `ch` using a send statement, `ch <- conn` (line 53). A server executes the receive expression `<-ch` (line 59), completing the communication. The channel was created without space to buffer values being sent (the default in Go), so after the 10 servers are busy with the first 10 connections, the eleventh `ch <- conn` will block until a server completes its call to serve and executes a new receive. The blocked communication operations create implicit back pressure on the listener, stopping it from accepting a new connection until it has handed off the previous one.

Note the lack of mutexes or other traditional synchronization mechanisms in these programs. Communication of data values on channels doubles as synchronization; by convention, sending data on a channel passes ownership from sender to receiver. Go has libraries that provide

mutexes, condition variables, semaphores, and atomic values for low-level uses, but a channel is often a better choice. In our experience, people reason more easily and more correctly about message passing—using communication to transfer ownership between goroutines—than they do about mutexes and condition variables. An early mantra was, "Do not communicate by sharing memory; instead, share memory by communicating."

Go's garbage collector greatly simplifies the design of concurrent APIs, removing questions about which goroutine is responsible for freeing shared data. As in most languages (but unlike Rust[22]), ownership of mutable data is not tracked statically by the type system. Instead, Go integrates with TSAN[28] to provide a dynamic race detector for testing and limited production use.

Security and Safety
Part of the reason for any new language is to address deficiencies of previous languages, which in Go's case included security issues affecting the safety of networked software. Go removes undefined behaviors that cause so many security problems in C and C++ programs. Integer types are not automatically coerced to one another. Null pointer dereferences and out-of-bounds array and slice indexes cause runtime exceptions. There are no dangling pointers into stack frames: Any variable that might possibly outlive its stack frame, such as one captured in a closure, will be moved to the heap instead. There are no dangling pointers in the heap either; the use of a garbage collector instead of manual memory management eliminates use-after-free bugs. Of course, Go doesn't fix everything, and there are things that were missed that perhaps should have been addressed. For instance, integer overflow could have been made a runtime error rather than defined to wrap around.

Since Go is a language for writing systems, which can require machine-level operations that break type safety, it is able to coerce pointers from one type to another and to perform address arithmetic, but only through the use of the `unsafe` package and its restricted special type `unsafe.Pointer`. Care must be taken to keep type-

system violations compatible with the garbage collector—for example, the garbage collector must always be able to identify whether a particular word is an integer or a pointer. In practice, the `unsafe` package appears very rarely: safe Go is reasonably efficient. Seeing `import "unsafe"` therefore serves as a signal to inspect a source file more carefully for possible safety problems.

Go's safety properties make it a much better fit for cryptographic and other security-critical code than a language such as C or C++. A trivial mistake, such as an out-of-bounds array index, which can lead to sensitive data disclosure or remote execution in C and C++, causes a run-time exception in Go, stopping the program and greatly limiting the potential impact. Go ships with a full suite of cryptography libraries, including SSL/TLS support; the standard library includes a production-ready HTTPS client and server. In fact, Go's combination of safety, performance, and high-quality libraries has made it a popular proving ground for modern security work. For example, the freely available certificate authority Let's Encrypt depends on Go for its production service[2] and recently crossed a milestone of one billion certificates issued.[1]

Completeness

Go provides the core pieces needed for modern development at the language, library, and tool levels. This requires a careful balance, adding enough to be useful "out of the box" while not adding so much that our own development processes bog down trying to support too many features.

The language provides strings, hash maps, and dynamically sized arrays as built-in, easily used data types. As noted earlier, these are sufficient for most Go programs. The result is greater interoperability between Go programs—for example, there are no competing implementations of strings or hash maps to fragment the package ecosystem. Go's inclusion of goroutines and channels is another form of completeness. These provide core concurrent functionality required in modern networked programs. Providing them directly in the language, as opposed to a library, makes it easier to tailor the syntax, the semantics, and the implementation to make them as lightweight and easy to use as possible, while providing a uniform approach for all users.

The standard library includes a production-ready HTTPS client and server. For programs that interact with other machines on the Internet, this is critical. Filling that need directly avoids additional fragmentation. We have already seen the `io.Writer` interface; any output data stream implements this interface by convention and interoperates with all other I/O adapters. Figure 1's `ListenAndServe` call, as another example, expects a second argument of type `http.Handler`, whose definition is shown in Figure 5. The argument `http.HandlerFunc(hello)`

Figure 3. A Go network server.

```
21  func listen(listener net.Listener) {
22      for {
23          conn, err := listener.Accept()
24          if err != nil {
25              log.Fatal(err)
26          }
27          go serve(conn)
28      }
29  }
30
31  func serve(conn net.Conn) {
32      for {
33          req, err := readRequest(conn)
34          if err != nil {
35              break
36          }
37          handle(req)
38      }
39  }
```

Figure 4. A Go network server, limited to 10 connections.

```
41  func listen(l net.Listener) {
42      ch := make(chan net.Conn)
43      const N = 10
44      for i := 0; i < N; i++ {
45          go server(ch)
46      }
47
48      for {
49          conn, err := l.Accept()
50          if err != nil {
51              log.Fatal(err)
52          }
53          ch <- conn
54      }
55  }
56
57  func server(ch chan net.Conn) {
58      for {
59          conn := <-ch
60          serve(conn)
61      }
62  }
```

Figure 5. The net/http package's handler interface.

```
type Handler interface {
    ServeHTTP(ResponseWriter, *Request)
}
```

implements its `ServeHTTP` method by calling `hello`. The library creates a new goroutine to handle each connection, as in the listener examples in this article's "Concurrency" section, so handlers can be written in a simple blocking style, and the server can scale automatically to handle many simultaneous connections.

The `http` package also provides a basic dispatcher, itself another implementation of `Handler`, which allows different handlers to be registered for different URL paths. Establishing `Handler` as the agreed-upon interface has enabled many different types of HTTP server middleware to be created and to interoperate. We did not need to add all these implementations to the standard library, but we did need to establish the interface that allows them to work together.

The standard Go distribution also provides integrated support for cross-compilation, testing, profiling, code coverage, fuzzing, and more. Testing is another area where establishing agreement about core concepts—such as what a test case is and how it is run—enabled the creation of custom testing libraries and test execution environments that all interoperate well.

Consistency

One goal we had for Go was for it to behave the same across different implementations, execution contexts, and even over time. This kind of "boring" consistent behavior allows developers to focus on their day-to-day work and allows Go to recede into the background.

To start, the language specifies consistent results as much as possible, even for erroneous behaviors such as null pointer dereferences and out-of-bounds array indexes, as discussed in this article's "Security and Safety" section. One exception where Go instead requires inconsistent behavior is iteration over hash maps. We found that programmers often inadvertently wrote code that depended on the hash function, causing different results on different architectures or Go implementations.

To make programs behave the same everywhere, one option would have been to mandate a specific hash function. Instead, Go defines that map iteration is non-deterministic. The im-

> **When we started designing Go, multicore computers were becoming widely available, but threads remained a heavyweight concept in all popular languages and operating systems.**

plementation uses a different random seed for each map and starts each iteration over a map at a random offset in the hash table. The result is that maps are consistently unpredictable across implementations: Code cannot accidentally depend on implementation details. In a similar vein, the race detector adds extra randomness to scheduling decisions, creating more opportunities to observe races.

Another aspect of consistency is performance over the lifetime of a program. The decision to implement Go using a traditional compiler, instead of the JIT used by languages such as Java and Node.js, provides consistent performance at startup time and for short-lived programs: There is no "slow start" penalizing the first few seconds of each process's lifetime. This quick startup has made Go an attractive target both for command-line tools, as noted in the previous section, and for scaled network servers such as Google App Engine.[30]

Consistent performance includes the overhead of garbage collection. The original Go prototype used a basic, stop-the-world garbage collector that, of course, introduced significant tail latency in network servers. Today, Go uses a fully concurrent garbage collector with pauses taking less than a millisecond,[21] and usually just a few microseconds, independent of heap size. The dominant delay is the time it takes the operating system to deliver a signal to a thread that must be interrupted.

A final kind of consistency is that of the language and libraries over time. For the first few years of Go's existence, we tinkered with and adjusted it in each weekly release. Users often had to change their programs when updating to a new Go version. Automated tools reduced the burden, but manual adjustments were also necessary. Starting with Go version 1, released in 2012, we publicly committed to making only backward-compatible changes to the language and standard library, so that programs would continue running unchanged when compiled with newer Go versions.[16] That commitment attracted industry and has encouraged not just long-lived engineering projects but also other efforts, such as books, training courses, and a thriving ecosystem of third-party packages.

Tool-Aided Development

Large-scale software development requires significant automation and tooling. From the start, Go was designed to encourage such tooling by making it easy to create.

A developer's daily experience of Go is through the `go` command. Unlike language commands that only compile or run code, the `go` command provides subcommands for all the critical parts of the development cycle: `go build` and `go install` build and install executables, `go test` runs test cases, and `go get` adds a new dependency. The `go` command also enables the creation of new tools by providing programmatic access to build details, such as the package graph.

One such tool is `go vet`, which performs incremental, package-at-a-time program analysis that can be cached the same way that caching compiled object files enables incremental builds. The `go vet` tool aims to identify common correctness problems with high precision, so that developers are conditioned to heed its reports. Simple examples include checking that formats and arguments match in calls to `fmt.Printf` and related functions, or diagnosing unused writes to variables or struct fields. These are not compiler errors, because we do not want old code to stop compiling simply because a new possible mistake has been identified. Nor are they compiler warnings; users learn to ignore those. Placing the checks in a separate tool allows them to be run at a time that is convenient for the developer, without interfering with the ordinary build process. It also makes the same checks available to all developers, even when using an alternate implementation of the Go compiler, such as Gccgo[15] or Gollvm.[17] The incremental approach makes these static checks efficient enough that we run them automatically during `go test`, before running the tests themselves. Testing is a time when users are looking for bugs anyway, and the reports often help explain actual test failures. This incremental framework is available for reuse by other tools as well.

Tooling that analyzes programs is helpful, but tooling that edits programs is even better, especially for program maintenance, much of which is tedious and ripe for automation.

The standard layout of a Go program is defined algorithmically. A tool, `gofmt`, parses a source file into an abstract syntax tree and then formats it back to source code using consistently applied layout rules.[14] In Go, it is considered a best practice to format code before storing it in source control. This approach enables thousands of developers to work on a shared codebase without the usual debates about brace styles and other details that accompany such large efforts. Even more significantly, tools can modify Go programs by operating on the abstract syntax form and then writing the result using `gofmt`'s printer. Only the parts actually changed are touched, resulting in "diffs" that match what a person would have arrived at by hand. People and programs can work together seamlessly in the same codebase.

To enable this approach, Go's grammar is designed to enable a source file to be parsed without type information or any other external inputs, and there is no preprocessor or other macro system. The Go standard library provides packages to allow tools to recreate the input and output sides of `gofmt`, along with a full type checker.

Before releasing Go version 1—the first stable Go release—we wrote a refactoring tool called `gofix`, which used these packages to parse the source, rewrite the tree, and write out well-formatted code. We used `gofix`, for example, when the syntax of deleting an entry from a map was changed. Each time users updated to a new release, they could run `gofix` on their source files to automatically apply the majority of the changes required to update to the new version.[5]

These techniques also apply to the construction of IDE plug-ins[29] and other tools—profilers, debuggers, analyzers, build automators, test frameworks, and so on—that support Go programmers. Go's regular syntax, the established algorithmic code-layout convention, and the direct standard library support make these kinds of tools much easier to build than they would otherwise be. As a result, the Go world has a rich, ever-expanding, and interoperating toolkit.

Libraries

After the language and tools, the next critical aspect of how users experience Go is the available libraries. As befits a language for distributed computing, in Go there is no central server where Go packages must be published. Instead, each import path beginning with a domain name is interpreted as a URL (with an implicit leading https://) giving the location of remote source code. For example, `import "github.com/google/uuid"` fetches code hosted in the corresponding GitHub repository.

The most common way to host source code is to point to a public Git or Mercurial server, but private servers are equally well supported, and authors have the option of publishing a static bundle of files rather than opening access to a source-control system. This flexible design and the ease of publishing libraries has created a thriving community of importable Go packages. Relying on domain names avoided a rush to claim valuable entries in a flat package name space.

It is not enough just to download packages; we must know which versions to use as well. Go groups packages into versioned units called modules. A module can specify a minimum required version for one of its dependencies, but no other constraints. When building a particular program, Go resolves competing required versions of a dependency module by selecting the maximum: If one part of the program requires version 1.2.0 of a dependency and another requires version 1.3.0, Go selects version 1.3.0—that is, Go requires the use of semantic versioning,[27] in which version 1.3.0 must be a drop-in replacement for 1.2.0. On the other hand, in that situation, Go will not select version 1.4.0 even when it becomes available, because no part of the program has asked explicitly for that newer version. This rule keeps builds repeatable and minimizes the potential risk of breakage caused by accidentally breaking changes introduced by new versions.

In semantic versioning, a module may introduce intentional breaking changes only in a new major version, such as 2.0.0. In Go, each major version starting at 2.0.0 is identified by a

major version suffix, such as /v2, in its import path: Distinct major versions are kept as separate as any other modules with different names. This approach disallows diamond dependency problems, and in practice it adapts to incompatibilities as well as systems with more finely grained constraints.[6]

To improve the reliability and reproducibility of builds downloading packages from all over the Internet, we run two services used by default in the Go toolchain: a public mirror of available Go packages and a cryptographically signed transparent log of their expected contents.[8,10,20] Even so, widespread use of software packages downloaded from the Internet continues to have security and other risks.[7] We are working on making the Go toolchain able to proactively identify and report vulnerable packages to users.

Conclusion

Although the design of most languages concentrates on innovations in syntax, semantics, or typing, Go is focused on the software development process itself. Go is efficient, easy to learn, and freely available, but we believe that what made it successful was the approach it took toward writing programs, particularly with multiple programmers working on a shared codebase. The principal unusual property of the language itself—concurrency—addressed problems that arose with the proliferation of multicore CPUs in the 2010s. But more significant was the early work that established fundamentals for packaging, dependencies, build, test, deployment, and other workaday tasks of the software development world, aspects that are not usually foremost in language design.

These ideas attracted like-minded developers who valued the result: easy concurrency, clear dependencies, scalable development and production, secure programs, simple deployment, automatic code formatting, tool-aided development, and more. Those early developers helped popularize Go and seeded the initial Go package ecosystem. They also drove the early growth of the language by, for example, porting the compiler and libraries to Windows and other operating systems (the original release supported only Linux and MacOS X).

Not everyone was a fan—for instance, some people objected to the way the language omitted common features such as inheritance and generic types. But Go's development-focused philosophy was intriguing and effective enough that the community thrived while maintaining the core principles that drove Go's existence in the first place. Thanks in large part to that community and the technology it has built, Go is now a significant component of the modern cloud computing environment.

Since Go version 1 was released, the language has been all but frozen. The tooling, however, has expanded dramatically, with better compilers, more powerful build and testing tools, and improved dependency management, not to mention a huge collection of open source tools that support Go. Still, change is coming: Go 1.18, released in March 2022, includes the first version of a true change to the language, one that has been widely requested—the first cut at parametric polymorphism. We left any form of generics out of the original language because we were keenly aware that it is very difficult to design well and, in other languages, too often a source of complexity rather than productivity. We considered a handful of designs during Go's first decade but only recently found one that we feel fits Go well. Making such a large language change while staying true to the principles of consistency, completeness, and community will be a severe test of the approach.

Acknowledgments

References
1. Aas, J. and Gran, S. Let's Encrypt has issued a billion certificates. Let's Encrypt (2020), https://letsencrypt.org/2020/02/27/one-billion-certs.html.
2. Aas, J., et al. Let's Encrypt: An automated certificate authority to encrypt the entire web. In *Proceedings of the 2019 ACM SIGSAC Conf. on Computer and Communications Security*, 2473–2487.
3. Bloch, D. Life on the edge: Monitoring and running a very large Perforce installation. Presented at *2007 Perforce User Conf.*, https://go.dev/s/bloch2007.
4. Chang, F., et al. Bigtable: A distributed storage system for structured data. In *7th USENIX Symposium on Operating Systems Design and Implementation* (2006), 205–218.
5. Cox, R. Introducing Gofix. *The Go Blog* (2011), https://go.dev/blog/introducing-gofix.
6. Cox, R. The principles of versioning in Go. (2019), https://research.swtch.com/vgo-principles.
7. Cox, R. Surviving software dependencies. *Communications of the ACM 62*, 9 (Aug. 2019), 36–43.
8. Cox, R. Transparent logs for skeptical clients (2019), https://research.swtch.com/tlog.
9. Cox, R. and Pike, R. Go programming. Presented at *Google I/O (2010)*, https://www.youtube.com/watch?v=jgVhBThJdXc.
10. Crosby, S.A. and Wallach, D.S. Efficient data structures for tamper-evident logging. In *Proceedings of the 18th USENIX Security Symp.* (2009), 317–334.
11. Donovan, A.A.A. and Kernighan, B.W. *The Go Programming Language*. Addison-Wesley, USA (2015).
12. Dorward, S., Pike, R., and Winterbottom, P. Programming in Limbo. In *IEEE COMPCON 97 Proceedings* (1997), 245–250.
13. Geissmann, L.B. Separate compilation in Modula-2 and the structure of the Modula-2 compiler on the personal computer Lilith. Ph.D. dissertation. Swiss Federal Institute of Technology (1983), https://www.cfbsoftware.com/modula2/ETH7286.pdf.
14. Gerrand, A. Go fmt your code. *The Go Blog* (2013), https://go.dev/blog/gofmt.
15. Go Project. Setting up and using gccgo. (2009), https://go.dev/doc/install/gccgo.
16. Go Project. Go 1 and the future of Go programs. (2012), https://go.dev/doc/go1compat.
17. Go Project. Gollvm, an LLVM-based Go compiler. (2017), https://go.googlesource.com/gollvm/.
18. Go Project. The Go programming language specification. (2021), https://go.dev/ref/spec.
19. Hoare, C.A.R. *Communicating Sequential Processes*. Prentice-Hall, Inc., USA (1985).
20. Hockman, K. Go Module Proxy: Life of a query. Presented at *GopherCon 2019*, https://www.youtube.com/watch?v=KqTySYYhPUE
21. Hudson, R.L. Getting to Go: The journey of Go's garbage collector. *The Go Blog* (2018), https://go.dev/blog/ismmkeynote.
22. Klabnik, S. and Nichols, C. *The Rust Programming Language*. No Starch Press, USA (2018).
23. Lam, A. Using remote cache service for Bazel. *Communications of the ACM 62*, 1 (Dec. 2018), 38–42.
24. Ousterhout, J. Why threads are a bad idea (for most purposes). (1995), https://web.stanford.edu/~ouster/cgi-bin/papers/threads.pdf
25. Pike, R. The implementation of Newsqueak. *Software: Practice and Experience 20*, 7 (1990), 649–659.
26. Pike, R., Dorward, S., Griesemer, R., and Quinlan, S. Interpreting the data: Parallel analysis with Sawzall. *Scientific Programming Journal 13* (2005), 277–298.
27. Preston-Werner, T. Semantic versioning 2.0.0. (2013), https://semver.org/
28. Serebryany, K., Potapenko, A., Iskhodzhanov, T., and Vyukov, D. Dynamic race detection with LLVM compiler: Compile-time instrumentation for ThreadSanitizer. In *Runtime Verification*, S. Khurshid, and K. Sen (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg (2012), 110–114.
29. Stambler, R. Go, pls stop breaking my editor. Presented at *GopherCon 2019*, https://www.youtube.com/watch?v=EFJfdWzBHwE.
30. Symonds, D., Tao, N., and Gerrand, A. Go and Google App Engine. *The Go Blog* (2011), https://go.dev/blog/appengine
31. Winterbottom, P. Alef language reference manual. In *Plan 9: Programmer's Manual Volume 2*. Harcourt Brace and Co., New York (1996).

Russ Cox (rsc@go.dev), **Robert Griesemer**, **Rob Pike**, **Ian Lance Taylor**, and **Ken Thompson** created the Go programming language and environment as software engineers at Google in Mountain View, California, USA. Cox, Griesemer, and Taylor continue to lead the Go project at Google, while Pike and Thompson have since retired.