Use Cases for Transparent Telemetry

Transparent Telemetry, Part 3

Russ Cox

February 8, 2023

research.swtch.com/telemetry-uses

I believe open-source software projects need to find an open-source-friendly way to do telemetry. This post is part of a short series of posts describing *transparent telemetry*, one possible answer to that challenge. For more about the rationale and background, see the introductory post*. For details about the design, see the previous post*.

For many years I believed we could make good enough decisions for Go without collecting any telemetry, by focusing on testing, benchmarks, and surveys. Over that time, however, I have collected many examples of decisions that can't be made in a principled way without better data, as well as performance problems that would go unnoticed. This post presents some of those examples.

What fraction of `go` command invocations still use GOPATH mode (as compared to Go modules)? What fraction of Go installations do?

Understanding how often GOPATH mode (non-module mode) is used is important for understanding how important it is to keep running, and for which use cases. Like many of these questions, knowing the basic usage statistics would help us decide whether or not to ask more specific questions on a Go survey or in research meetings with Go users. We are also considering changes like fine-grained compatibility (#56986* and #57001*) and per-iteration loop scoping (#56010*) that depend heavily on version information available only in module mode. Understanding how often GOPATH mode is used would help us understand how many users those changes would leave behind.

To answer these questions, we would need two counters: P, the number of times the `go` command is invoked in GOPATH mode, and N, the the number of times the `go` command is invoked at all. The answer to the first question is P divided by N. When collected across all sampled systems, it wouldn't make sense to present the total P divided by the total N. Instead, we'd want to present the distribution of P/N in sampled reports, plotting the cumulative distribution of P/N over all systems. The answer to the second question is the number of reports with $P > 0$ divided by the total number of reports. This can also be read off the cumulative distribution graph by looking for $P/N > 0$.

What fraction of `go` command invocations use Go workspaces (a `go.work` file)? What fraction of Go installations do?

We added Go workspaces in Go 1.18. Most of the time, `go.work` files will be most appropriate in modules that are not dependencies of other modules, such as closed-source projects. Understanding how often workspaces are used in practice would help us prioritize work on them as well as understand whether they are as useful as we expected. That knowledge would again inform future survey questions and research interviews.

Answering these questions requires the N counter from above as well as a new counter for the number of times the `go` command is invoked in a Go workspace.

What fraction of go command invocations or Go installations use `-buildmode=shared`? What fraction of Go installations do?

As mentioned in the introductory post*, `-buildmode=shared` has never worked particularly well, and its current design is essentially incompatible with Go modules. We have a rough design* for how we might make it work, but that's a lot of work, and it's unclear whether it should be a high priority. Our guess is that there are essentially no users of this flag, and that therefore it shouldn't be prioritized. But that's just a guess; data would be better.

Answering these questions requires the N counter as well as a new counter for the number of times the go command is invoked with `-buildmode=shared`. In general we would probably want to define a counter for the usage of every command flag. When the set of flag values is limited to a fixed set, as it is for `-buildmode`, we would also include the flag value, to distinguish, for example, `-buildmode=c-shared` (which is working and easy to support) from `-buildmode=shared` (which is not). As noted in the discussion of this example in the main text, it is possible that only certain build machines using `-buildmode=shared` and those all also have telemetry disabled. If so, we won't see them. This is a fundamental limitation of telemetry: it serves as an additional input into decisions, not as the only deciding factor.

What fraction of go command invocations use a specific GOOS, GOARCH, or subarchitecture setting? What fraction of Go installations do?

Go has a porting policy* that distinguishes between first-class and secondary ports. The first-class ports are the ones that the Go team at Google supports directly and that are important enough to stop a release if a serious bug is encountered. We do not have a principled, data-driven way to decide which ports are first-class and which are secondary. Instead, we make guesses based on prior knowledge and other non-Go usage statistics. Today the first-class ports are Linux, macOS, and Windows running on 386, amd64, arm, and arm64. Should we add FreeBSD to the list? (Maybe?) What about RISC-V? (Probably not yet?) At the subarchitecture level, is it time to retire support for `GOARM=5` (ARMv5), which does not have support for atomic memory operations in hardware and must rely on the operating system kernel to provide them? (I have no idea.) Data would be helpful in making those decisions.

Answering these questions requires the N counter as well as a new counter for the number of times each of the Go-specific environment variables takes one of their known values. For example, there would be different counters for the go command running with `GOARCH=amd64`, `GOARCH=arm`, `GOOS=linux`, `GOOS=windows`, `GOARM=5`, `GO386=sse2`, and so on.

What fraction of Go installations run on a particular major version of an operating system such as Windows 7, Windows 8, or WSL?

As operating systems get older, they often become harder to support. It gets harder to run them in modern virtual machines for testing, kernel bugs affecting Go are left unfixed, and so on. One of the considerations* when deciding to end support for a port is the amount of usage it gets, yet we have no reliable way to measure that. Recently we decided to end support for Windows 7 (#57003*) and Windows 8 (#57004*). We were able to rely on other considerations, namely Microsoft's own support policy, as well as operating system statistics collected by companies deploying Go binaries. It would be helpful to have our own direct statistics as well. More recently, we announced that Go 1.20 will be the last release to support macOS High Sierra and were promptly asked to keep it around (#57125*).

Answering these questions would require counters for each major operating system version like `Windows 7`, `Windows 8`, `Debian Buster`, `Linux 3.2.x`, and so on. There would be no need to collect fine-grained information like specific patch releases, unless some specific release was important for some reason (for example, we might want more resolution on Linux 2.6.x, to inform advancing the minimum Linux kernel version beyond 2.6.32).

What fraction of builds cross-compile, and what are the most common cross-compilation targets?

Go has always been portable and supported cross-compilation*. That's clearly important and won't be removed, but what are the host/target pairs that are most important to make work well? For example, Go 1.20 will improve compiling macOS binaries from a Linux system: the binaries will now use the host DNS resolver, same as native macOS builds. After I made this change, multiple users reached out privately to thank me for solving a major pain point for them. This was a surprise to me: I was unaware that cross-compiling macOS binaries from Linux was so common. Are there other common cross-compilation targets that merit special attention?

Answering these questions uses the same counters as in the last section, along with information about the native default `GOOS` and `GOARCH` settings.

What is the overall cache miss rate in the Go build cache?

The Go build cache is a critical part of the user experience, but we don't know how well it works in practice. The original build cache heuristics were decided in 2017 based on a tracing facility that we asked users to run for a few weeks* and submit traces. Since then, many things about Go builds have changed, and we have no idea how well things are working.

Answering this question requires a counter for build cache misses and another for build cache hits or accesses.

What is the typical cache miss rate in the Go build cache for a given build?

A more detailed question would be what a typical cache miss rate is for a given build. We hope it would be very low, and if it went up, we'd want to investigate why. This is something that users are unlikely to notice and report directly, but it would still indicate a serious problem worth fixing.

Answering this question would require keeping an in-memory total of cache misses and hits, and then at the end of the build computing the overall rate and incrementing a counter corresponding to one column of a histogram, as described in the introductory post*.

What fraction of `go` command invocations recompile the standard library?

The cache hit rate for standard library packages should be very high, since users are not editing those packages. Before Go 1.20, distributions shipped with pre-compiled `.a` files that should have served as cache hits for default builds. Only cross-compilation or changing build tags or compiler flags should have caused a rebuild, and then those would have been cached too. The cache hit rate in the standard library can therefore serve as a good indicator that the build cache is working properly. Especially if the overall hit rate is low, checking the standard library hit rate should help distinguish a few possible failure modes. As noted at the introductory post*, this metric would have detected a bug invalidating the pre-compiled macOS `.a` files that lingered for six releases instead.

Answering this question would require keeping a counter of the number of `go` command invocations that recompile any standard library package, along with the total number of invocations N.

What is the typical cache hit age in the Go build cache?

Another important question is how big the cache needs to be. Today the build cache has a fixed policy of keeping entries for 5 days after last use and then deleting them, based on the traces we collected in 2017. Some users have reported the build cache being far too large for their machines (#29561*). One possible change would be to reduce the maximum time since last use. Perhaps 1 day or 3 days would be almost as effective, at a fraction of the cost.

Answering this question would require making another histogram of counters, this time for a cache hit in an entry with a given age range.

Would a generational build cache work effectively?

Another possible change to the build cache would be to adopt a collection policy inspired by generational garbage collection*: an entry is deleted after a few hours, unless it is used a second time, in which case it is kept until it hasn't been used again for a few days. Would this work?

Answering this question would require a second histogram counting the age distribution of cache entries at their first reuse, as well as a counter of the number of cache entries deleted without any reuse at all.

How many modules are listed in a typical Go workspace?

Go 1.18 workspaces allow working in multiple work modules simultaneously. How many work modules are in a typical workspace? We don't know, so we don't know how important per-module overheads are.

Answering this question would require a histogram of workspace size.

How many `go.mod` files are loaded in a typical `go` command invocation?

In addition to the one or more modules making up the workspace, every time the `go` command starts up it needs to load information about all the dependency modules. In many projects, the number of direct module requirements appears small but indirect requirements add many more modules. What is a typical number of modules the `go` command must load?

Go 1.17 introduced module graph pruning, which lists all used transitive dependencies in the top-level `go.mod` file for a project and then avoids reading `go.mod` files from unused indirect dependencies. That is reduces the number of `go.mod` files loaded. We know it works on our test cases and a few real repositories like Kubernetes, but we don't know how well it works across a variety of Go installations. If we'd had telemetry before the change, we should have seen a significant drop in this statistic. If not, that would be a bug to investigate.

Answering this question would require a histogram of the number of modules loaded.

What is the distribution of time spent downloading modules in a given build?

Even with module graph pruning, the `go` command still spends time downloading `go.mod` files and module content into the cache as needed. How much time is spent on downloads? If it's a lot, then maybe we should work on reducing that number, such as by parallelizing downloads or improving download speed.

Answering this question would require a histogram of time spent in the module download phase of the `go` command.

What is the latency distribution of a Go workspace load?

Most `go` command invocations will not need to download anything. Those are bottlenecked by the overall workspace load, which spends its time reading the

`go.mod` files and scanning all the Go packages to understand the import graph and relevant source files. How long does that typically take?

We found out from discussions with users that the number was significantly higher in module mode than in GOPATH mode in a real system, so in Go 1.19 we introduced module cache indexing, which lets the `go` command avoid reading the top of every Go source file in every dependency at startup to learn the import graph. If we'd had telemetry before the change, we should have seen a significant drop in this statistic. If not, that would be a bug to investigate.

Answering this question would require a histogram of time spent in the workspace load phase of the `go` command.

What is the module cache index miss rate?

The module cache index should have a very low miss rate, because changes to the module's dependency graph are much less common than changes to source files. Is that true on real systems? If not, there's a bug to find.

Answering this question would require computing the overall module cache index miss rate during a build and then keeping a histogram of rates observed, just like for the build cache miss rate.

What fraction of builds encounter corrupt Go source files in the module cache? Does this percentage correlate with host operating system?

We have some reports of corrupted Go source files when using the module index (#54651*). This corruption seems very uncommon, and it seems to manifest only when using `GOOS=linux` under the Windows Linux subsystem WSL2, suggesting that the problem may not be Go's fault at all. There is not an obvious path forward for that issue, and we don't understand the severity, although it appears to be low. If we could change the Go toolchain to work around observed corruption but also self-report its occurrence, then we could get a better sense of the impact of this bug and how important it is to fix. For a bug like this, we would typically reason that it is not happening very often so it would be better to leave it open and learn more about what causes it than to add a workaround and lose that signal. Telemetry lets us both add the workaround and watch to see that the bug is not a sign of a larger problem.

Answering this question would require having a counter for encountering invalid Go source files in the module cache. That should be a rare event, and if we saw an uptick we could look at correlation with other information like the default `GOOS` and the operating system major versions, provided WSL2 was reported as something other than Linux.

How much CPU and RAM does a typical invocation of the compiler or linker take?

We have invested significant effort into reducing the CPU and RAM requirements of both the compiler and the linker. The linker in particular was almost completely reengineered in Go 1.15 and Go 1.16. We have some benchmarks to exercise them, but we don't know how that translates into real-world performance. The linker work was motivated by performance limits in Google's build environment compiling Google's Go source code. Other environments may have significantly different performance characteristics. If so, that would be good to know.

Answering this question would require a CPU histogram for the compiler, a RAM histogram for the compiler, and then CPU and RAM histograms for the linker as well.

What is the relative usage of different versions of the Go toolchain?

The current Go release support policy* is that each release is supported until there are two newer major releases. With work like the Go compatibility policy*, we aim to make it easy to update to newer releases, so that users do not get stuck on unsupported releases. How well is that working? We don't know. The fine-grained compatibility work (#56986* and #57001*) aims to make it easier to update to a newer toolchain, which should mean usage of older versions falls off quicker. It would be helpful to confirm that with data. Also, once there is a baseline for how quickly new versions are adopted, seeing one lag would indicate a problem worth investigating. For example, I recently learned about a Windows timer problem (#44343*) that is keeping some Windows users on Go 1.15. If we saw that in the data, we could look for what is holding users back.

Answering this question would not require any direct counters. The reports are already annotated with which version of Go was running, so the percentage can be calculated as number of reports from a specific version of Go divided by total reports.

What is the relative distribution of Go versions in go lines in the work module during a build?

Today the `go` line indicates roughly (but not exactly) the maximum version of Go that a module targets. Part of proposal #57001* is to change it to be the exact minimum permitted version of Go. It would be good to know how often modules set that line to the latest version of Go as compared to the previous version of Go or an unsupported earlier version, for much the same reasons as the previous question.

Answering this question requires a counter for each released `go` version, incremented for each build using a `go` line with that version. Another way to answer this question would be to scan open-source Go modules, but conventions in the open-source Go library ecosystem may well be different from conventions in the production systems that use those packages. We don't want to overfit our decisions to what happens in open-source code.

What fraction of builds encounter `//go:debug` lines? What is the relative usage of each setting?

The fine-grained compatibility work (#56986*) proposes to add a new `//go:debug` line used in main packages to set default `GODEBUG` settings. Those settings would be guaranteed to be provided for two years, but after that we would be able to retire settings as needed. Knowing how much each one is used would let us make informed decisions about whether to retire specific settings.

Answering these questions requires a counter for each `GODEBUG` setting, incremented for each build using a `//go:debug` line with that setting. As before, another possibility would be to scan open-source Go modules, but the vast majority of open source is libraries, not main packages, so there would be very little signal, and the signal we do get could be very different from the reality of production usage.

What fraction of builds use C code via cgo? What about C++, Objective-C, Fortran, or SWIG?

Cgo is another area where the open-source Go library ecosystem is almost certainly not aligned with actual production usage: cgo tends to be discouraged in most public Go libraries but is a critical part of interoperation with existing code in many production systems. We don't know how many though, nor what the cgo use looks like. Go added support for C++ and SWIG early on, due to usage at Google. Objective-C and Fortran were added at user request. Objective-

C probably gets lots of use now on macOS and iOS, but what about Fortran? Is anyone using it at all? Do we need to prioritize testing it or fixing bugs in it?

Answering these questions requires a counter for each `go` build of a package using cgo, and then additional counters for when the cgo code includes each of the languages of interest. Again we could scan open-source Go modules, but production usage is likely to be different from open-source libraries.

What fraction of builds with cgo code explicitly disable cgo? What fraction implicitly disable it? What fraction of builds implicitly disable cgo because there is no host C compiler?

Many builds insist on not using cgo by setting `CGO_ENABLED=0`. Cgo is also disabled by default when cross-compiling. Starting in Go 1.20, cgo is disabled by default when there is no host C compiler. It would be helpful to know how often these cases happen. If lots of builds explicitly disable cgo, that would be something to ask about in surveys and research interviews. If lots of builds implicitly disable cgo due to cross-compiling, we might want to look into improving support for cgo when cross-compiling. If lots of builds implicitly disable cgo for not having a host C compiler, we might want to talk to users to understand how it happens and then update documentation appropriately.

Answering these questions requires counters for builds with cgo code, builds with an explicit `CGO_ENABLED=0`, builds with implicit disabling due to cross-compilation, and builds with implicit disabling due to lack of host C compiler.

What fraction of Go installations use a particular version of a Go toolchain dependency such as Clang, GCC, Git, or SWIG?

The Go toolchain contains many workarounds for old versions of tools that it invokes, like Clang, GCC, Git, and SWIG. For example, Clang 3.8's `-g` flag is incompatible with `.note.GNU-stack` sections, so the `go` command has to look for a specific error and retry without the flag. The `go` command also invokes various `git` commands to understand the repository it is working inside, and it is difficult to tell whether a new command will be safe. For example, `go` uses the global `git -c` option, which was added in Git 1.7.2. Is that old enough to depend on? We decided yes in 2018, but that broke CentOS 6. Even today we have not resolved what versions we should require for version control dependencies (#26746*). Data would help us make the decision.

Answering these questions requires counters for builds that invoke each of these tools along with per-tool-version counters, giving a version histogram for each tool.

What fraction of builds use `GOEXPERIMENT=boringcrypto`?

Go published a fork using BoringCrypto* years ago; in Go 1.19 the code moved into the main tree, accessible with `GOEXPERIMENT=boringcrypto`. The code is unsupported today, but if there was substantial usage, that might make us think more about whether to start supporting it.

Answering this question requires a counter for builds with `GOEXPERIMENT=boringcrypto`.

What are the observed internal compiler errors in practice?

Because compilers try to keep processing a program even after finding the first error, it is common to see index out of bounds or nil reference panics happen in code that has already been diagnosed as invalid. Instead of crashing in this situation, the Go compiler has a panic handler that checks whether any errors have already been printed. If so, then the compiler silently exits early (unsuccessfully), leaving the user to fix the reported errors and try again. After the report-

ed errors are fixed, the next run is likely to work just fine. Hiding these panics provides a much better experience to users, but each such panic is still a bug we would like to fix. Since the panics are successfully masked, users will never file bug reports for them. It would be helpful to collect stack traces for these panics, even when the compiler exits gracefully.

Answering this question would require a crash counter (corresponding to the current call stack) to be incremented in the panic handler even when the panic is otherwise silenced.

What is the relative distribution of compiler errors printed during development?

The errors printed by the compiler for invalid programs are a critical part of the overall Go user experience. In the early days of Go, I spent a lot of time making sure that the errors printed for my own buggy programs were clear. But different Go users make different mistakes: helping new Go users with their own programs was an important way I found out about other errors that needed to be clearer, Today, Go usage has grown far beyond anything we can understand from personal experience. Understanding which errors people hit the most would help us decide which ones to focus on in research interviews and make clearer or more specific. Commonly encountered errors might also prompt ideas about language changes. For example, we made unused imports and variables errors before `go vet` existed. If users hit those frequently enough to be bothersome, we might consider making them no longer compiler errors and instead leave them to be `go vet` errors.

It would also be interesting to compare the error distribution from the compiler against the error distribution from `gopls` for users who work in an IDE. Does the IDE make certain errors less (or more) likely? Does it keep code with certain errors from ever reaching the compiler?

Answering these questions would require a counter for each distinct error message printed. One option would be to have each error-reporting call site in the compiler also declare a counter passed to the error-reporting function. Another option would be to treat a call to the error-reporting function itself as a crash, incrementing a crash counter corresponding to the current call stack, perhaps truncated to just a few frames.

What is the relative distribution of `go vet`-reported errors printed during development?

The errors printed by `go vet` are its entire user experience. They have to be good. Just like with the compiler, we don't know which ones are printed the most and might need more attention. If there are `vet` errors that are never or rarely printed, that would also be good to know. Perhaps the checks are broken, or perhaps they are no longer necessary and can be removed. As a concrete example, we noticed recently that the vet shift check has unfortunate false positives (#58030*). Data about how often the check triggers would be a useful input to the decision about what to do about it.

Answering this question would use the same approach as in the compiler, either explicit counters or short-stack crash counters.

What fraction of Go installations invoke `go vet` explicitly, as opposed to the automatic run during `go test`?

I suspect that the majority of Go users never run `go vet` themselves, that they only run it implicitly as part of `go test`. Am I right? If so, we should probably spend more time improving the analyses that were deemed too noisy for `go test`, because those aren't running and therefore aren't helping find bugs.

Answering this question would require a counter for `go vet` invocations and a counter for `go test`-initiated `vet` invocations.

What is the relative distribution of specific analyses run using `go vet`?

When `go vet` is run explicitly, passing no command-line flags runs all analyses. Adding command-line flags runs only specific analyses. Do most runs use all analyses or only specific ones? If the latter, which ones? If we found that some analyses were being avoided, we might want to do further research to understand why. If we found that some analyses were being run very frequently, we might want to look into adding it to the `go test` set.

Answering this question would require a counter for `go vet` invocations and a counter for each analysis mode.

What is the relative usage of different versions of `gopls`?

`Gopls` is the only widely-used Go tool maintained by the Go team that ships separate from the Go distribution. As with Go releases, knowing which versions are still commonly used would help us understand which still need to be supported. Knowing how frequently versions are updated would help us tell when the upgrade process needs to be easier or more well documented.

Answering this question would require a counter incremented at `gopls` startup, since reports already separate counter sets by the version of the program being run.

What fraction of `gopls` installations connect to specific editors?

`Gopls` is an LSP server* that can connect to any editor with LSP support. Different editors speak LSP differently, though, and editor-specific bugs are common. If we knew which editors are most commonly used, we could prioritize those in testing.

Answering this question would require a counter for `gopls` being invoked from each known editor (Emacs, Vim, VS Code, and so on), along with a counter for "Other", since we cannot collect counters with arbitrary values. If "Other" became a significant percentage, a survey or user research interview would help us identify a new, more specific counter to add.

Which settings is `gopls` configured with?

`Gopls` has many settings*, some of which are deprecated or obsolete. We don't know which ones can be removed without disrupting users, or how many users would be disrupted by each (#52897*, #54180*, #55268*, #57329*). We guess as best we can, but real data would be better.

Answering this question would require a counter for each setting, incremented at `gopls` startup or after each setting change. Like with the LSP editors and the compiler flags, only a fixed collection of settings can be collected. For settings that take arbitrary user-defined values, there can only be a counter for the setting being used at all.

What fraction of `gopls` installations make use of a given code lens?

`Gopls` provides quite a few code lenses* that users can invoke using their editor UI. Which ones actually get used? If one isn't being used, we would want to use a survey or user research interview to understand why and either improve or remove it.

Answering this question would require a counter for each code lens setting being enabled, incremented at `gopls` startup or after each configuration change.

What fraction of `gopls` installations disable or enable a given analyzer?

`Gopls` provides quite a few analyzers* as well. These run while a program is being edited and show warnings or other information about the code. If many users turn off an analyzer that is on by default, that's a useful signal that the analyzer has too many false positives or is too noisy in some other way. And if many users enable an analyzer that is off by default, that's a useful signal that we should consider turning it on by default instead.

Answering this question would require a counter for each analyzer being enabled, incremented at `gopls` startup or after each configuration change.

What fraction of `gopls` code completion suggestions are accepted?

One core feature of `gopls` is making code completion suggestions. Are they any good? We only know what we see from our own usage and from limited user research studies, which are time-consuming to collect, cannot be repeated frequently, and may not be representative of Go users overall. If instead we knew what fraction of suggestions were accepted, we could tell whether suggestions need more work and whether they are getting better or worse.

Answering this question would require a counter for the number of suggestions made as well as the number of times the first suggestion was accepted, the number of times the second suggestion was accepted, and so on.

What is the latency distribution for important `gopls` editor operations, like saving a file, suggesting a completion, or finding a definition?

We believe that performance greatly influences overall user happiness with `gopls`: when we spent a few months in 2022 working on performance improvements, user-reported happiness in VS Code Go surveys noticeably increased. We can run synthetic performance benchmarks, but there is no substitute for actual real-world performance measurements.

Answering this question would require a histogram for each key operation.

How often does `gopls` crash? Why?

`Gopls` is a long-running server that can, over time, find its way into a bad state and crash. We spent a few months in 2022 on overall `gopls` stability improvements, and we believe the crash rate is much lower now, but we have no real-world data to back that up.

Answering this question would require counters for each time `gopls` starts and each time it crashes. Each crash should also increment a crash counter to record the stack.

How often does `gopls` get manually restarted?

Sometimes `gopls` hasn't crashed but also isn't behaving as it should, so users manually restart it. It is difficult to know why `gopls` is being restarted each time, but at the least we can track how often that happens and then follow up in sureys or user research interviews to understand the root cause.

Answering this question would require counters for each time `gopls` starts and each time it is manually restarted.

Is the body of this `if` statement ever reached?

In general, when refactoring or cleaning up code it is common to see a workaround or special case and wonder if it ever happens anymore and how.

Answering this question would require a stack counter incremented at the start of the body of the `if` statement in question. The reports would then include not just the fact that the body is reached but also a few frames of context explaining the call stack to it to help understand how it is reached.

And so on ...

I could keep going, but I will stop here. It should be clear that knowing the answers to these questions really would help developers working on Go to make better, more informed decisions about feature work and identify performance bugs and other latent problems in Go itself. It should also be clear from the breadth of examples that there is no way to ask all these questions on a survey, at least not one that users will finish. Many of them, including all the histograms, are completely impractical as survey questions anyway.

I hope it is also clear that the basic primitive of counters for predefined events is powerful enough to answer all these questions, while at the same time not exposing any user data, path names, project names, identifiers, or anything of that sort.

Finally, I hope it is clear that none of this is terribly specific to Go. Any open source project with more than a few users has the problem of understanding how the software is used and how well it's working.

For more background about telemetry and why it is important, see the introductory post*. For details about the design, see the previous post*.

* Asterisks mark hyperlinked text.