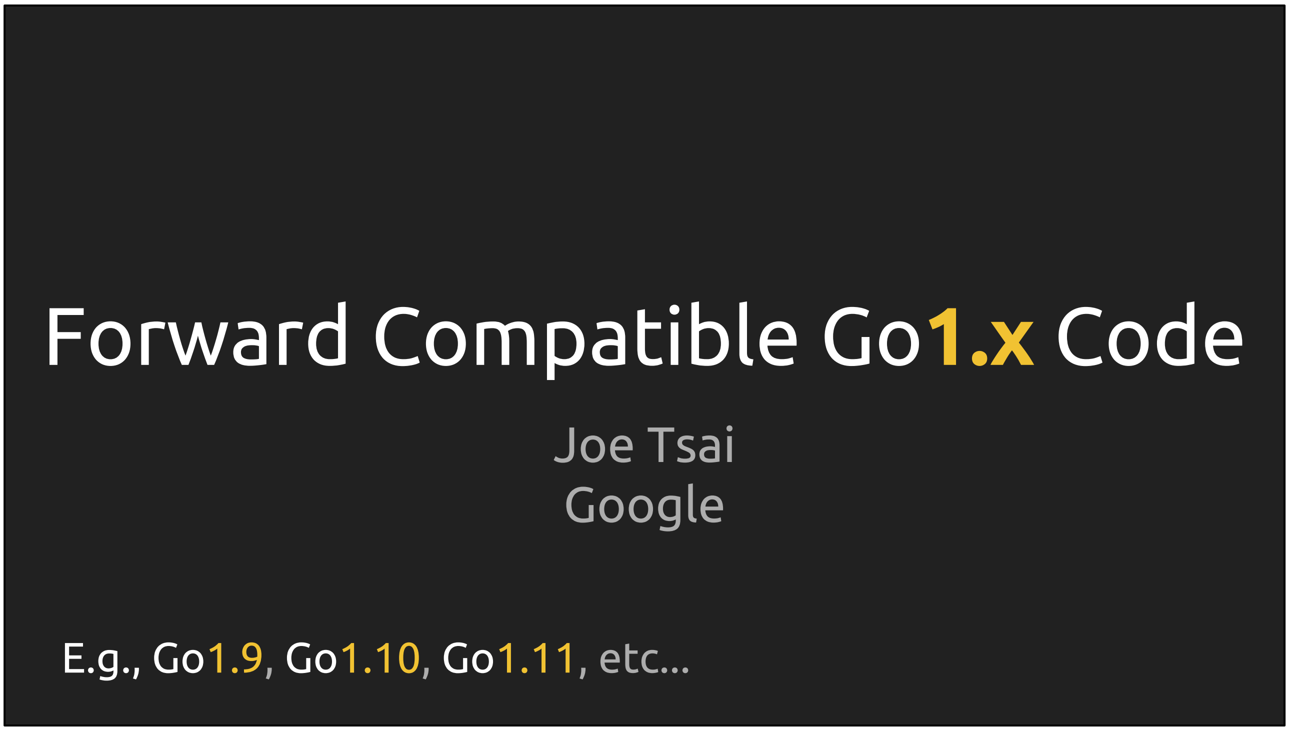
Forward Compatible Go Code

Joe Tsai
Google

Hi, good evening.

As mentioned, my name is Joe, and I work on the Go team in Sunnyvale on libraries for building Google infrastructure.
I want to thank you all for the opportunity today to talk about Forward Compatible Go Code.



Forward Compatible Go1.x Code

Joe Tsai
Google

E.g., Go1.9, Go1.10, Go1.11, etc...

Now, I know Russ just started the conversation about Go2,
so I want to be explicit that this talk is about
compatibility within the Go1.X series of releases (e.g., 1.9, 1.10, 1.11, etc...),
but the same basic principles covered today still apply in Go2.

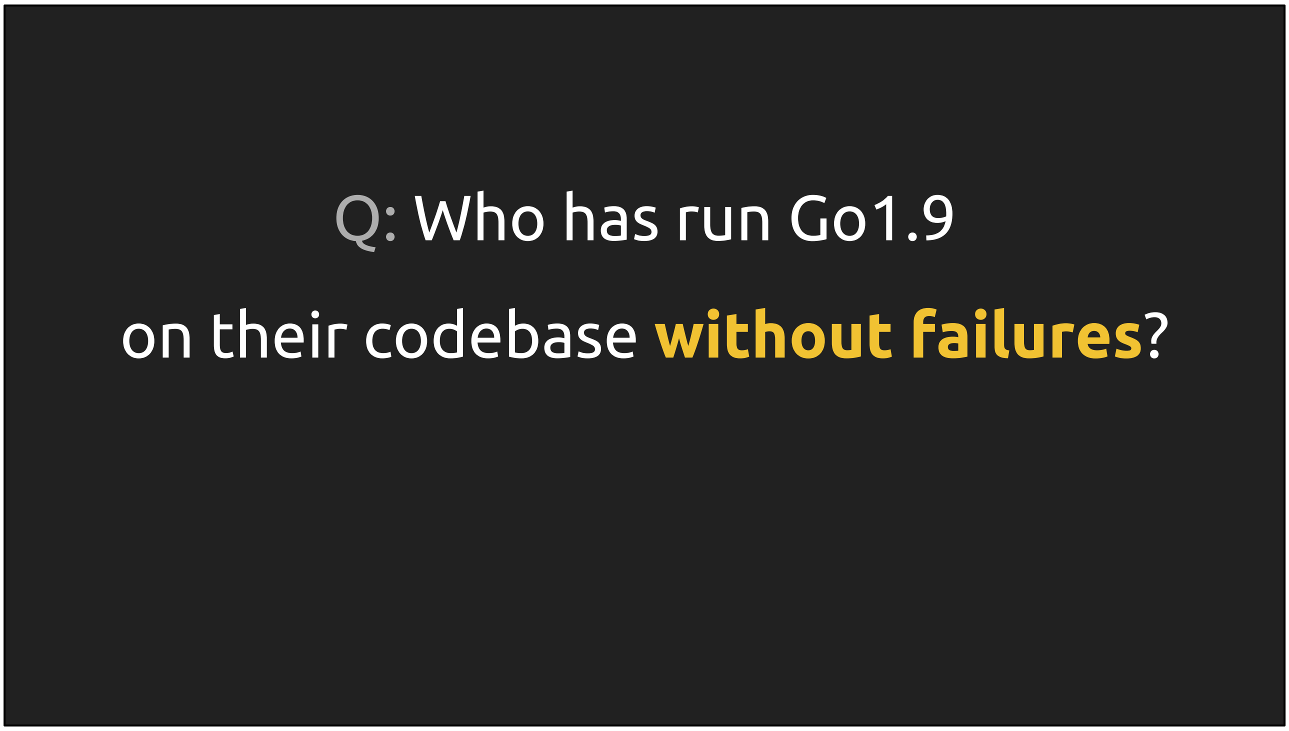
So. Before I get started, who’s excited for Go1.9!?

Q: Who has run Go1.9 on their codebase?

Yea! Alright, can you please raise your hands
if you have tried running Go1.9 on your codebase?

(PAUSE: Wait for hands to go up)

Okay! Great! That’s a fair number of you!



Q: Who has run Go1.9 on their codebase **without failures**?

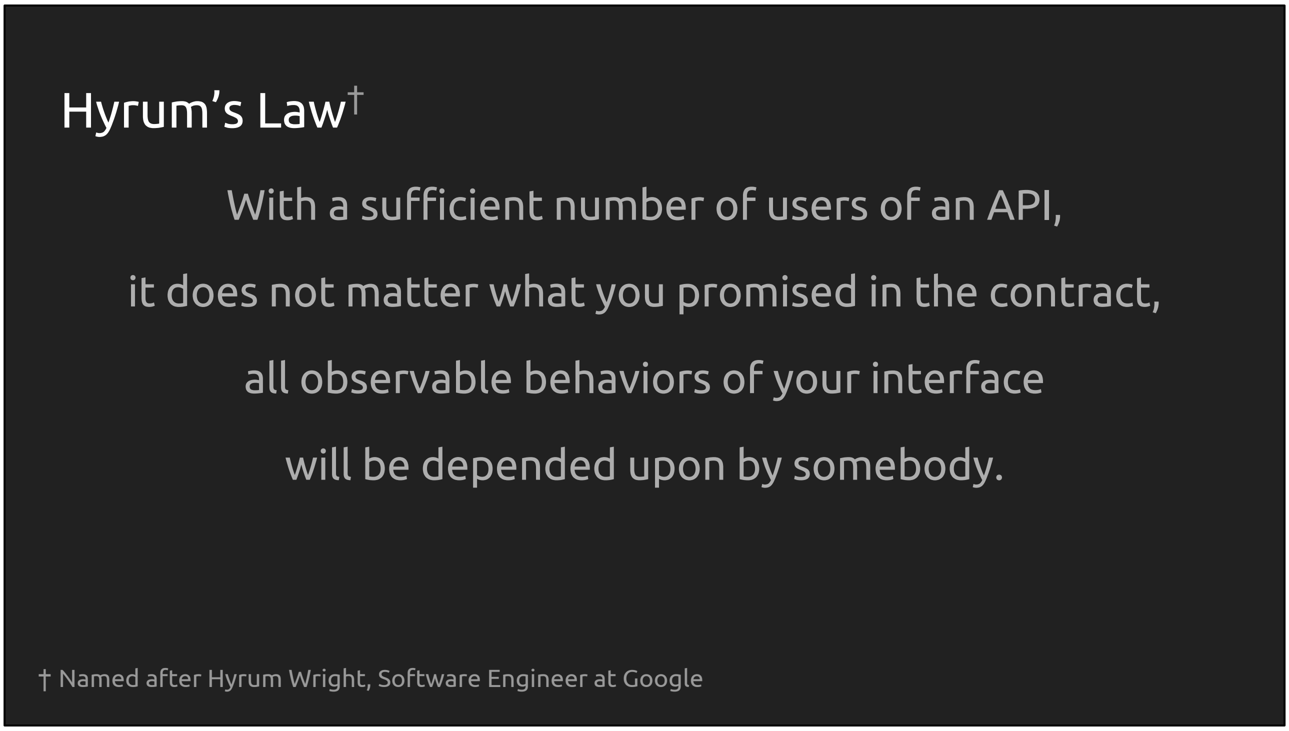
Here's a follow-up question.
Keep your hands raised if
    you successfully ran Go1.9 without any failures
    (and lower them if you did encounter failures).

(PAUSE: Wait for hands to go down)

Okay… it seems fewer of you were able to upgrade without any issues.

So, what's going on here?
Some important goals for Go are stability and development scalability.
In theory, the team of world-wide contributors developing Go
    should be able to improve the toolchain
    without breaking the code of those who depend on it.
However, in practice, as we see here today, that is not always the case.

What's going on is an observation that we, at Google, call Hyrum's Law.



Hyrum's Law[†]

With a sufficient number of users of an API,

it does not matter what you promised in the contract,

all observable behaviors of your interface

will be depended upon by somebody.

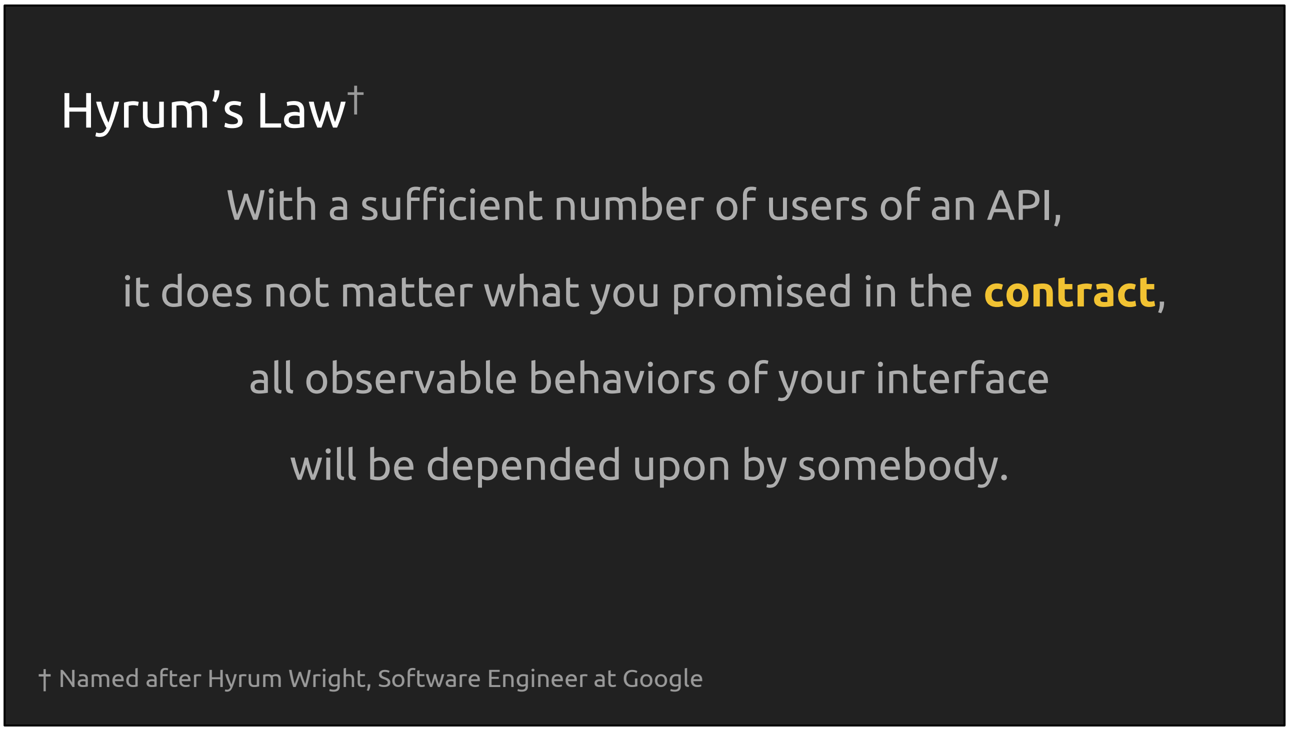
† Named after Hyrum Wright, Software Engineer at Google

Hyrum's Law reads:

With a sufficient number of users of an API,
it does not matter what you promised in the contracts,
all observable behaviors of your interface
will be depended upon by somebody.

You see, when breakages occur trying to run Go1.9 on your codebase,
it means that something somewhere in your code
is depending on some behavior specific to 1.8
that no longer behaves the exact same way in 1.9.
These differences in behavior alter the control flow of your program
resulting in bugs and test failures.

In Hyrum's law, I want to highlight the word contract.



Hyrum's Law[†]

With a sufficient number of users of an API,

it does not matter what you promised in the **contract**,

all observable behaviors of your interface

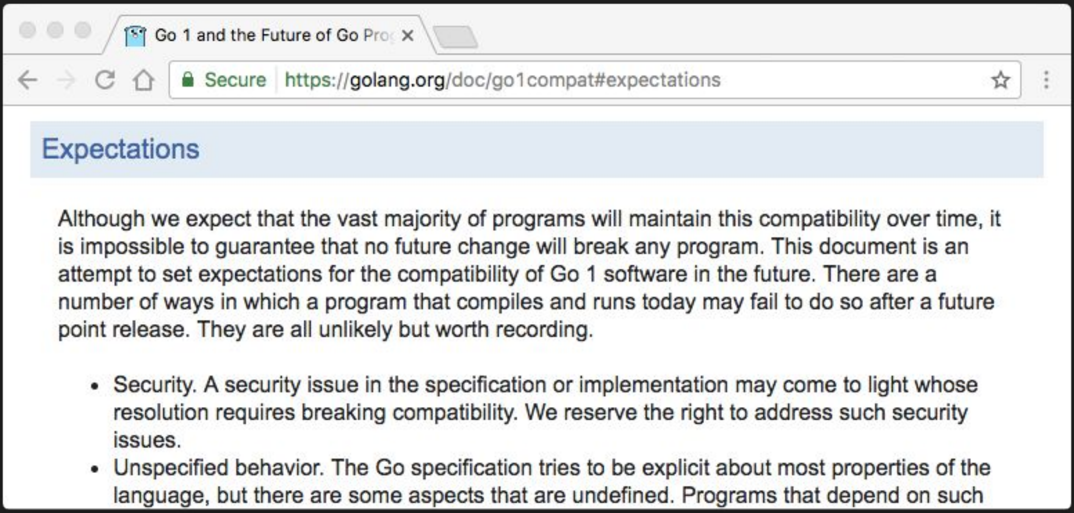
will be depended upon by somebody.

† Named after Hyrum Wright, Software Engineer at Google

The contract is the set of exported package identifiers and associated documentation
that promises what an API will do for you.
It specifies the guaranteed behaviors that your program may rely upon.
But if you depend on unspecified behavior,
you are treading in murky waters.

To ensure that future versions of Go continue to run old Go code,
we have our own contract,
which is commonly known as the "Go1 Compatibility Promise".

The Go1 Compatibility Promise


Go 1 and the Future of Go Pro
Secure
https://golang.org/doc/go1compat#expectations

Expectations

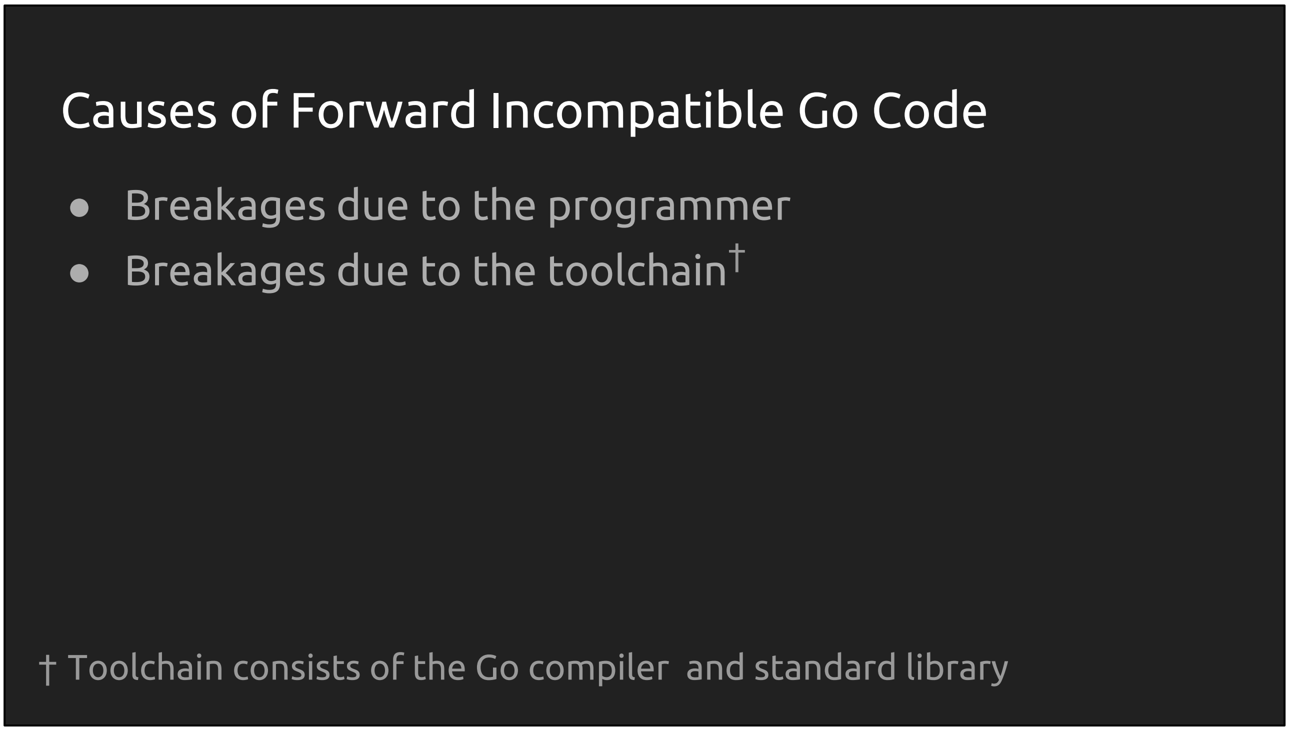
Although we expect that the vast majority of programs will maintain this compatibility over time, it is impossible to guarantee that no future change will break any program. This document is an attempt to set expectations for the compatibility of Go 1 software in the future. There are a number of ways in which a program that compiles and runs today may fail to do so after a future point release. They are all unlikely but worth recording.

- Security. A security issue in the specification or implementation may come to light whose resolution requires breaking compatibility. We reserve the right to address such security issues.
- Unspecified behavior. The Go specification tries to be explicit about most properties of the language, but there are some aspects that are undefined. Programs that depend on such

https://golang.org/doc/go1compat

The compatibility promise covers a wide range of topics including
security, bugs, use of struct literals and methods, use of unsafe, etc.
We don’t have time to cover every aspect of the compatibility promise,
so I encourage you to read the entire document online.
Fundamentally, it was designed with the goal of creating a stable platform for developing Go projects.
It’s an agreement between the users of Go and the authors of the Go toolchain
(i.e., the Go language specification and the standard library).
If this agreement is met, then in theory,
all Go1 programs should continue to compile and run correctly, unchanged, on all future Go1 releases.

In practice, however, some of you do run into issues using newer releases of Go.
This comes down to two primary causes.

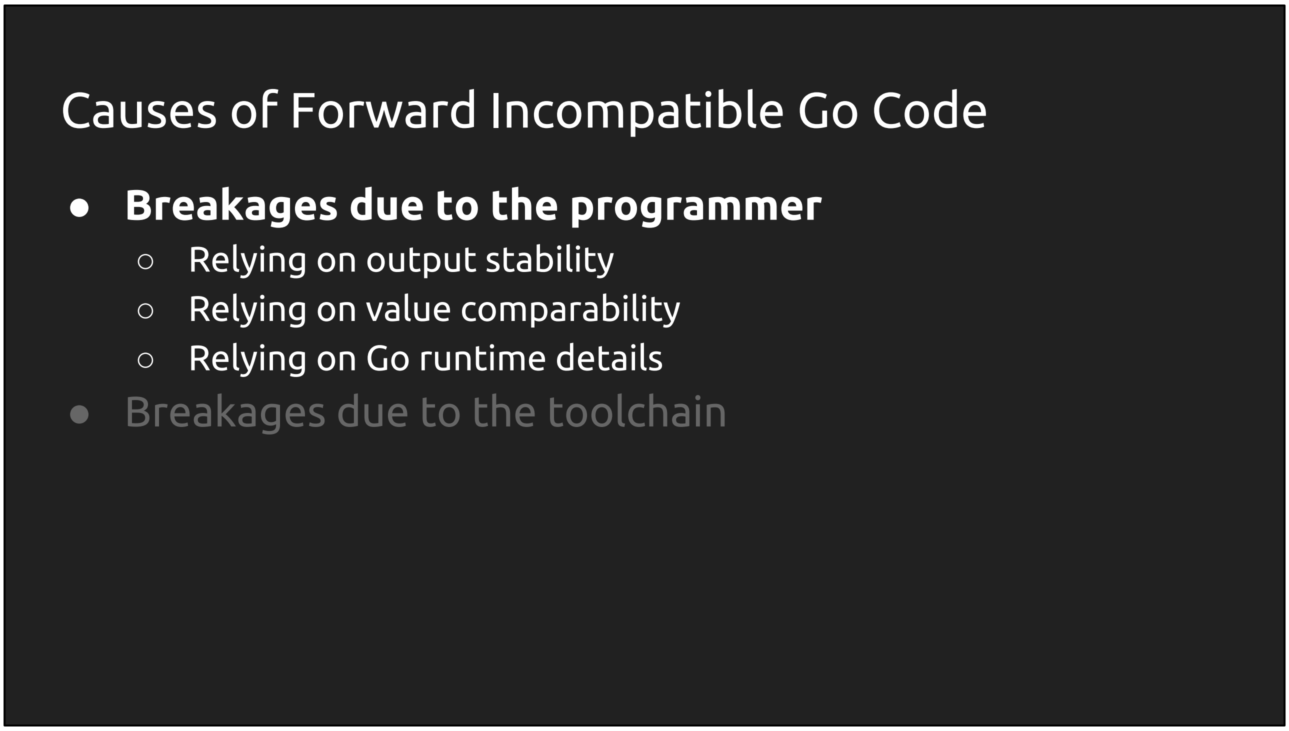


Causes of Forward Incompatible Go Code

- Breakages due to the programmer
- Breakages due to the toolchain†

† Toolchain consists of the Go compiler and standard library

The first major cause of breakages is the Go programmer
relying on behavior outside the compatibility agreement.
The second major cause of breakages are faults in the toolchain,
where the Go compiler and/or standard library changed promised behavior.
One or both of these causes may be involved in any specific breakage.

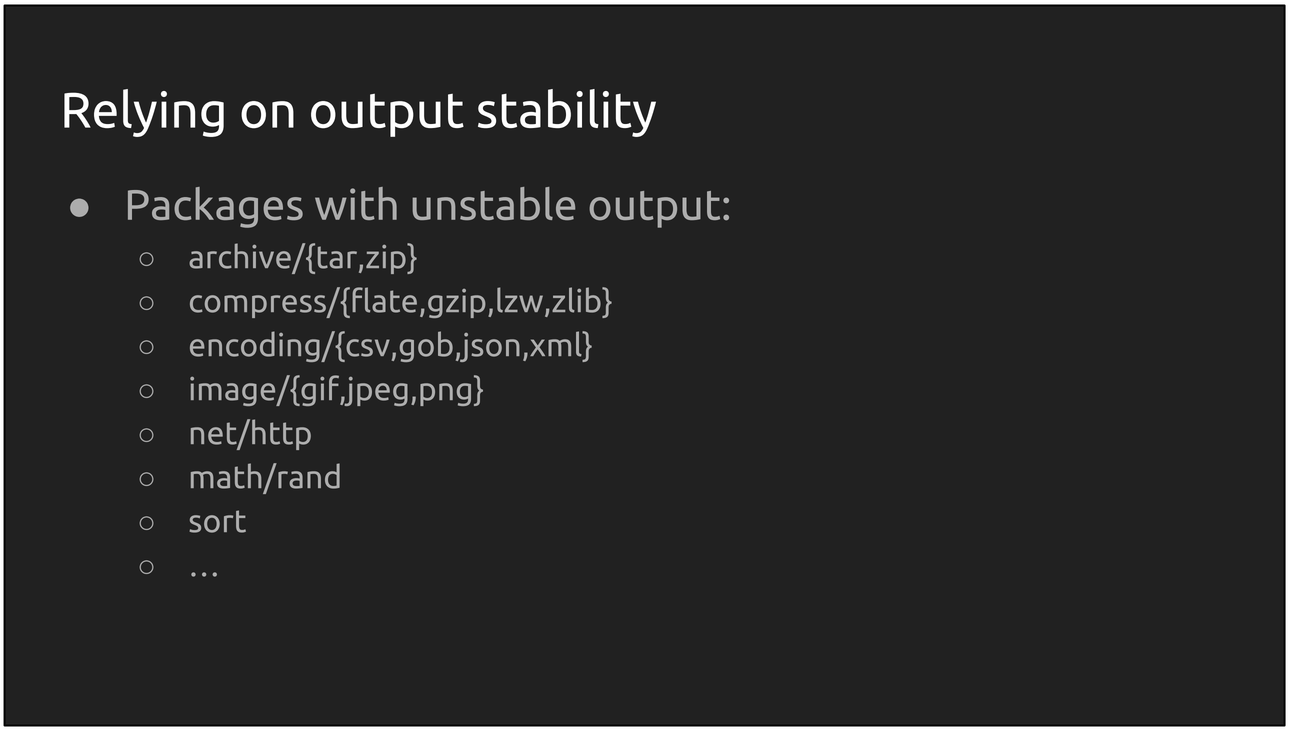


Causes of Forward Incompatible Go Code

- **Breakages due to the programmer**
  - Relying on output stability
  - Relying on value comparability
  - Relying on Go runtime details
- Breakages due to the toolchain

Let us focus first on breakages due to the programmer.
A vast majority of the time, code is brittle because
it unintentionally depends on unspecified behavior.

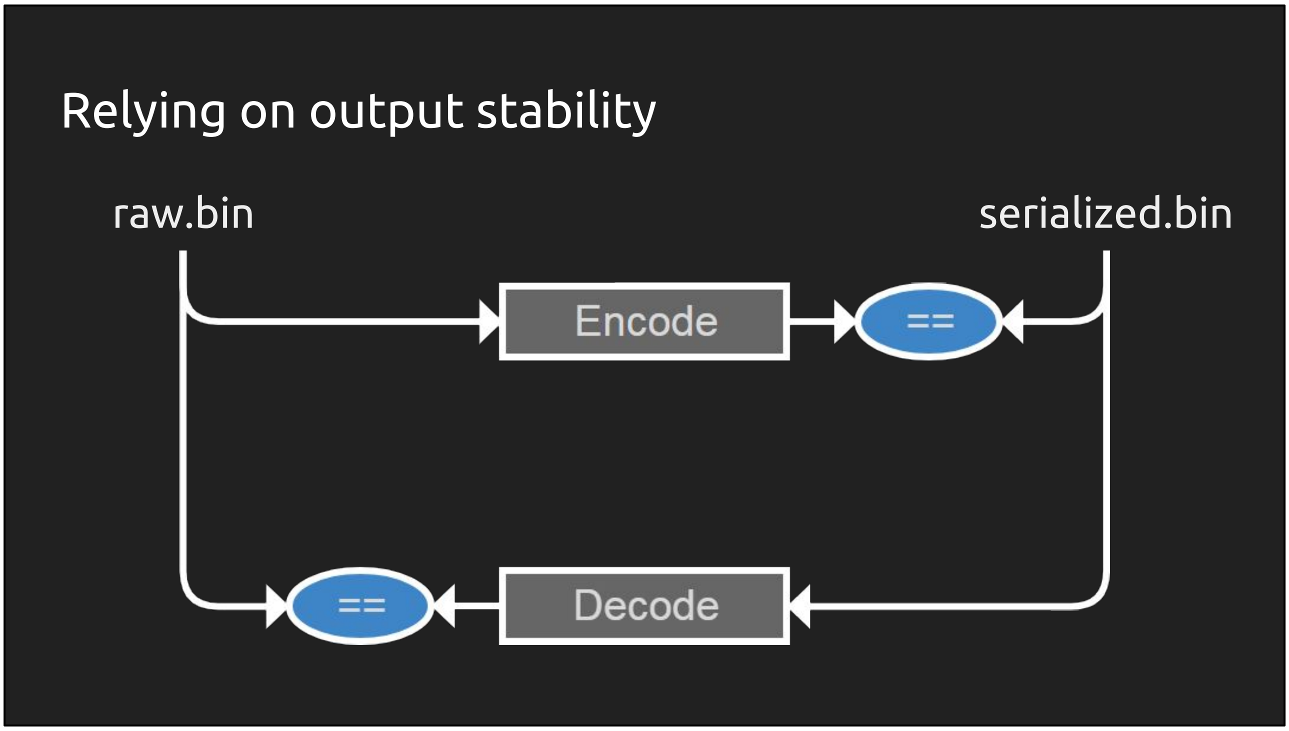
As part of my job, I deploy the latest Go toolchain across all Go code at Google.
From this experience, I want to share three categories of erroneous assumptions
that I've seen repeatedly lead to brittle and buggy code,
in the hopes that you can avoid making similar mistakes.



Relying on output stability

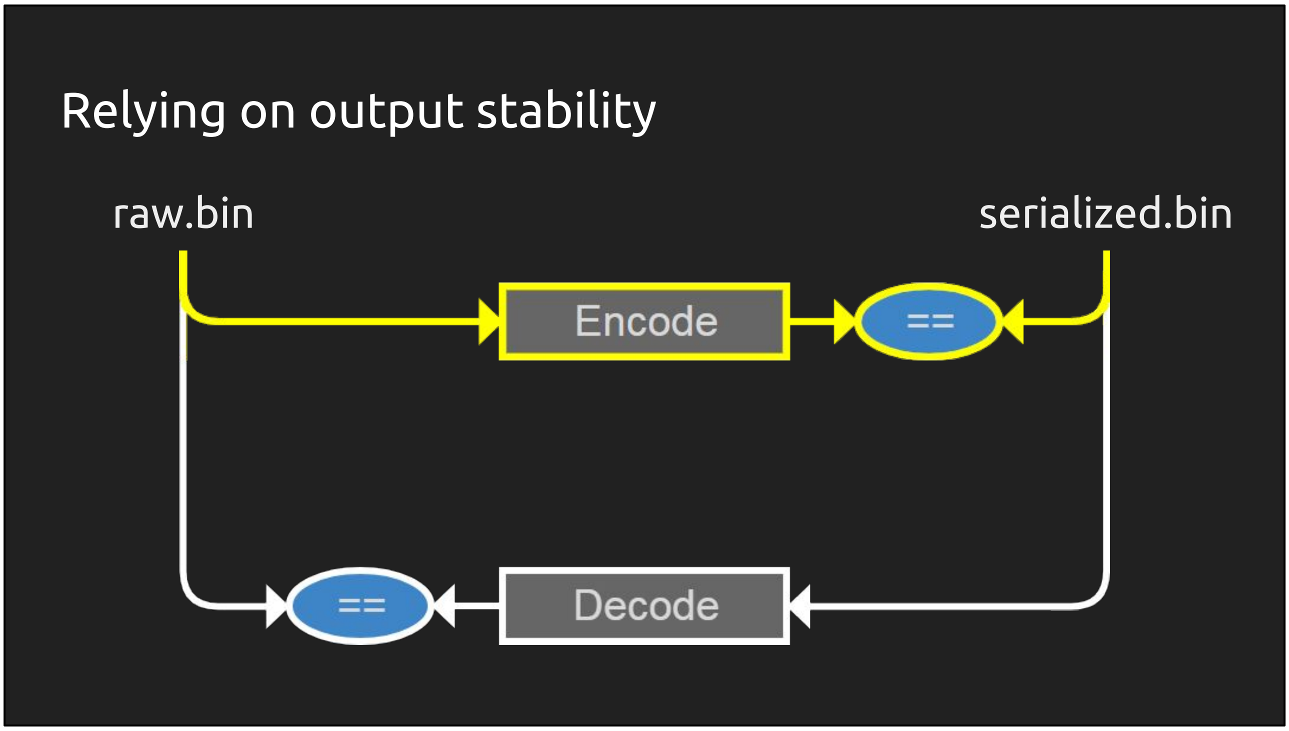
- Packages with unstable output:
  - archive/{tar,zip}
  - compress/{flate,gzip,lzw,zlib}
  - encoding/{csv,gob,json,xml}
  - image/{gif,jpeg,png}
  - net/http
  - math/rand
  - sort
  - ...

The first category of erroneous assumptions is
that the output of a package will be stable from release-to-release.
Here we a have a list of packages
that users often incorrectly assume to be stable.
Each of these packages does guarantee
that the output is compliant with the targeted specification
(e.g., the json package outputs valid JSON).
However, having unstable output means that
it is not safe to assume that the outputs are byte-for-byte identical.


Relying on output stability

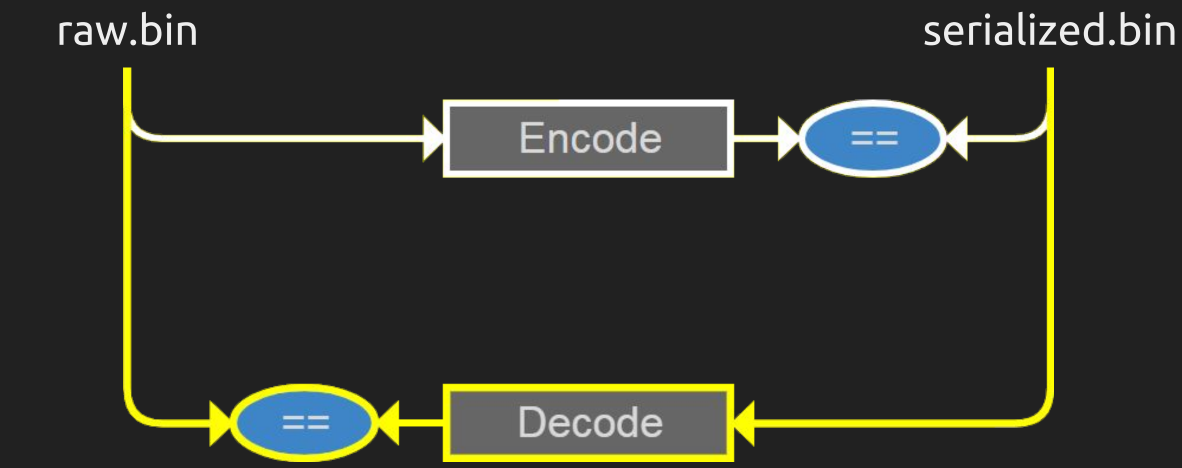
This often fails in tests that are written like this.

Imagine that you wrote a serialization package
that performs some type of encoding and decoding.
As part of its serialization process,
your package depends on the json package.


Relying on output stability

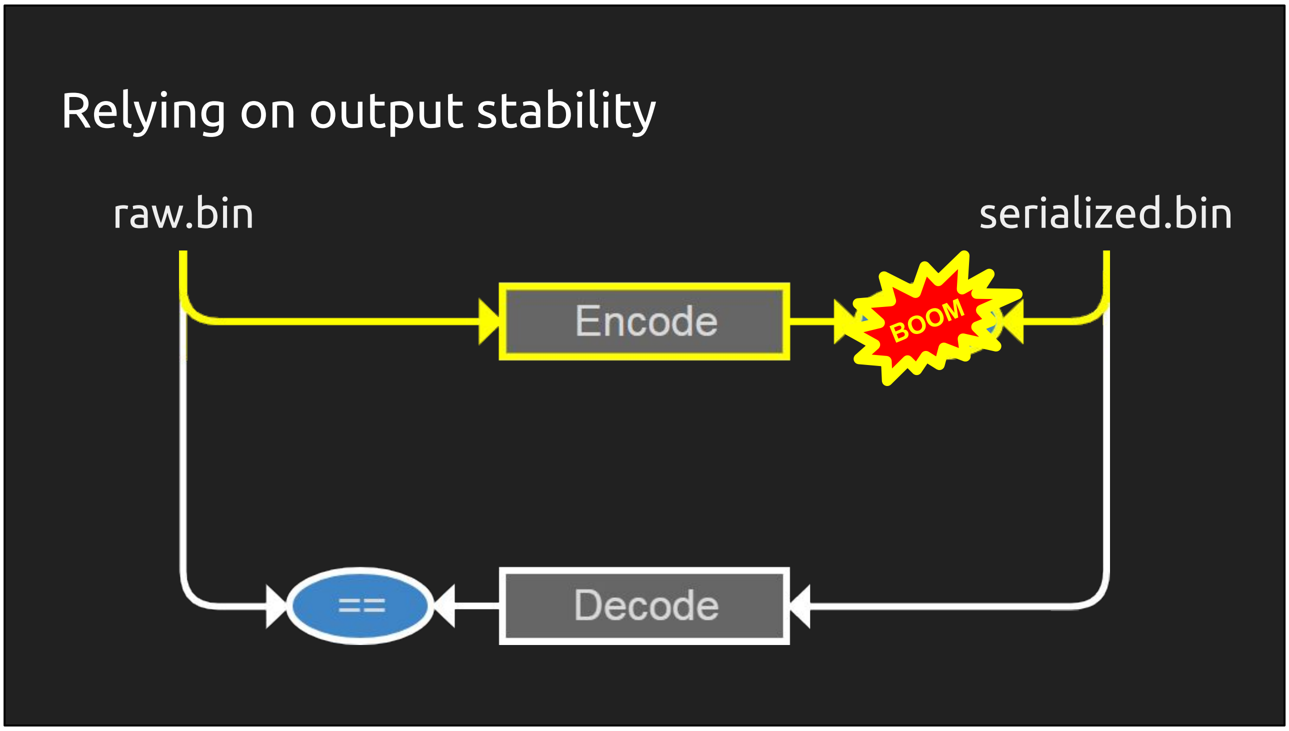
When testing your package functions,
you might want to pass some hard-coded raw test data into Encode and check that the result exactly matches some hard-coded pre-serialized test data.

Relying on output stability



Similarly, you might pass that same serialized data into Decode and check that the decoded value exactly matches the raw data.

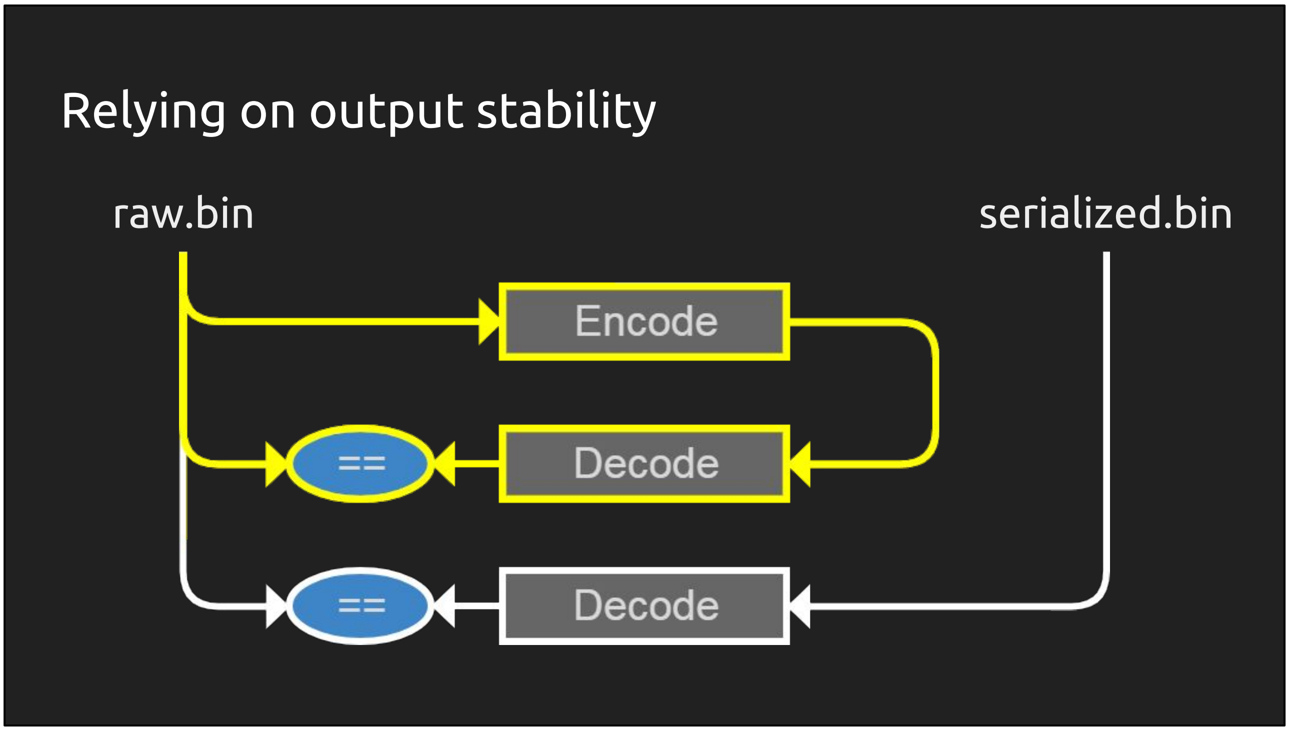
At first glance, this seems reasonable. However, this is problematic...



Relying on output stability

When you upgrade to a newer version of Go,
the JSON package may now output something slightly different
causing your test to fail the byte-for-byte comparison.

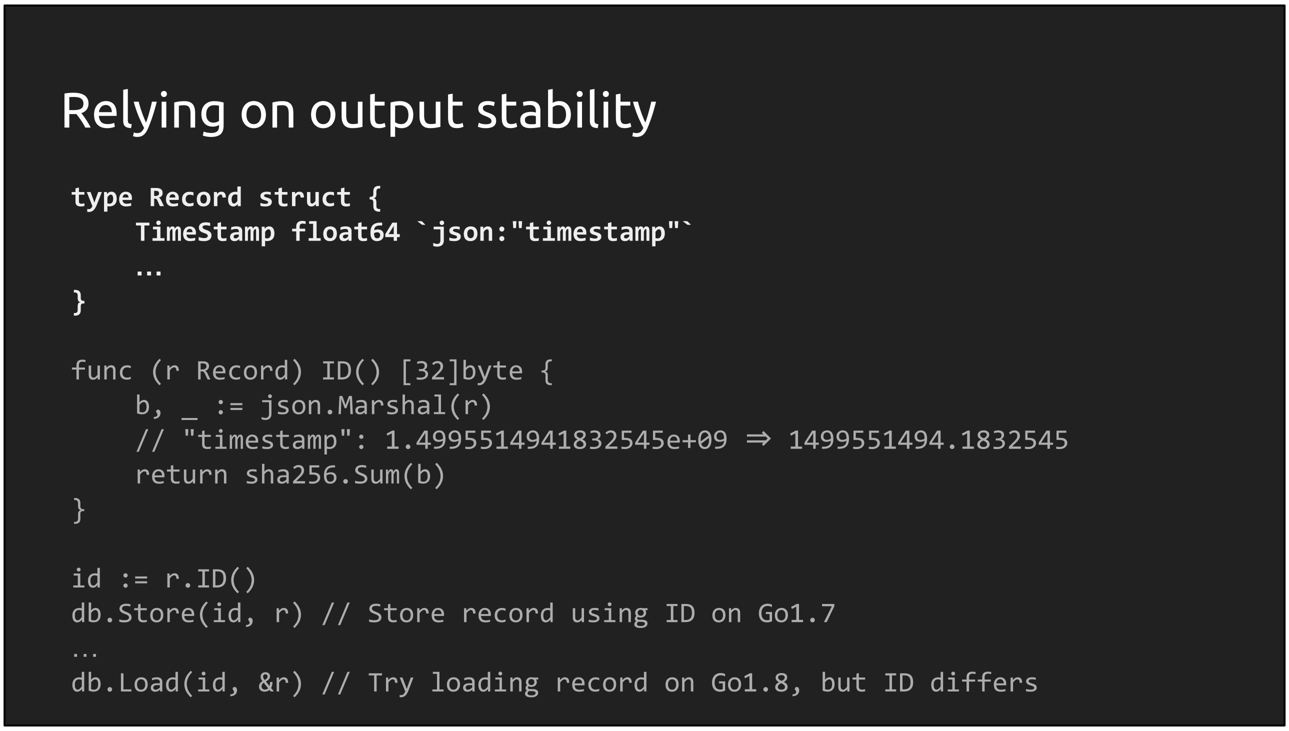
Instead, a better way to structure this tests is to...



Relying on output stability

...roundtrip the encoder back into the decoder and ensure that you retrieve the same input.
This approach avoids relying on the json package's output being stable,
and only relies on the json package being able to decode valid JSON.

Even more serious than broken tests,
I have seen this cause bugs production code...



Relying on output stability

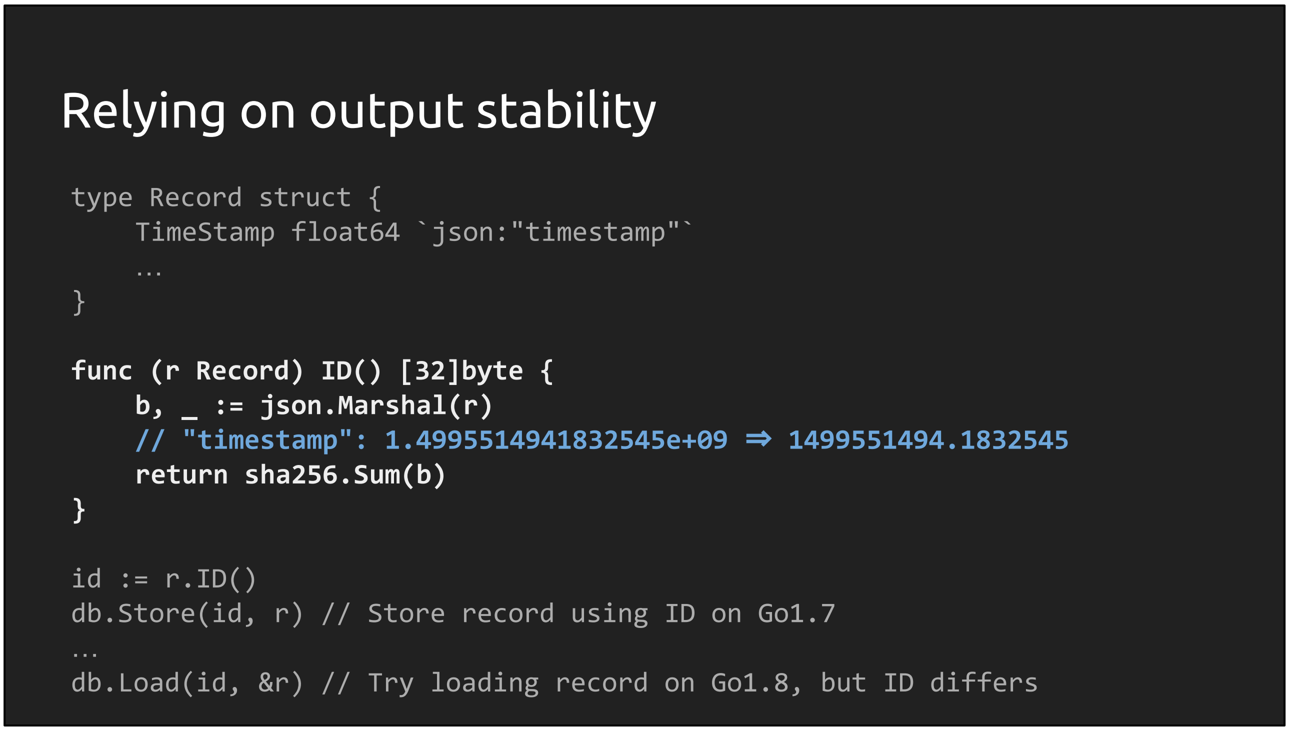
```
type Record struct {
    TimeStamp float64 `json:"timestamp"`
    …
}

func (r Record) ID() [32]byte {
    b, _ := json.Marshal(r)
    // "timestamp": 1.4995514941832545e+09 ⇒ 1499551494.1832545
    return sha256.Sum(b)
}

id := r.ID()
db.Store(id, r) // Store record using ID on Go1.7
…
db.Load(id, &r) // Try loading record on Go1.8, but ID differs
```

Such as this example.

Here we have a type Record where one of the fields is a TimeStamp in seconds.



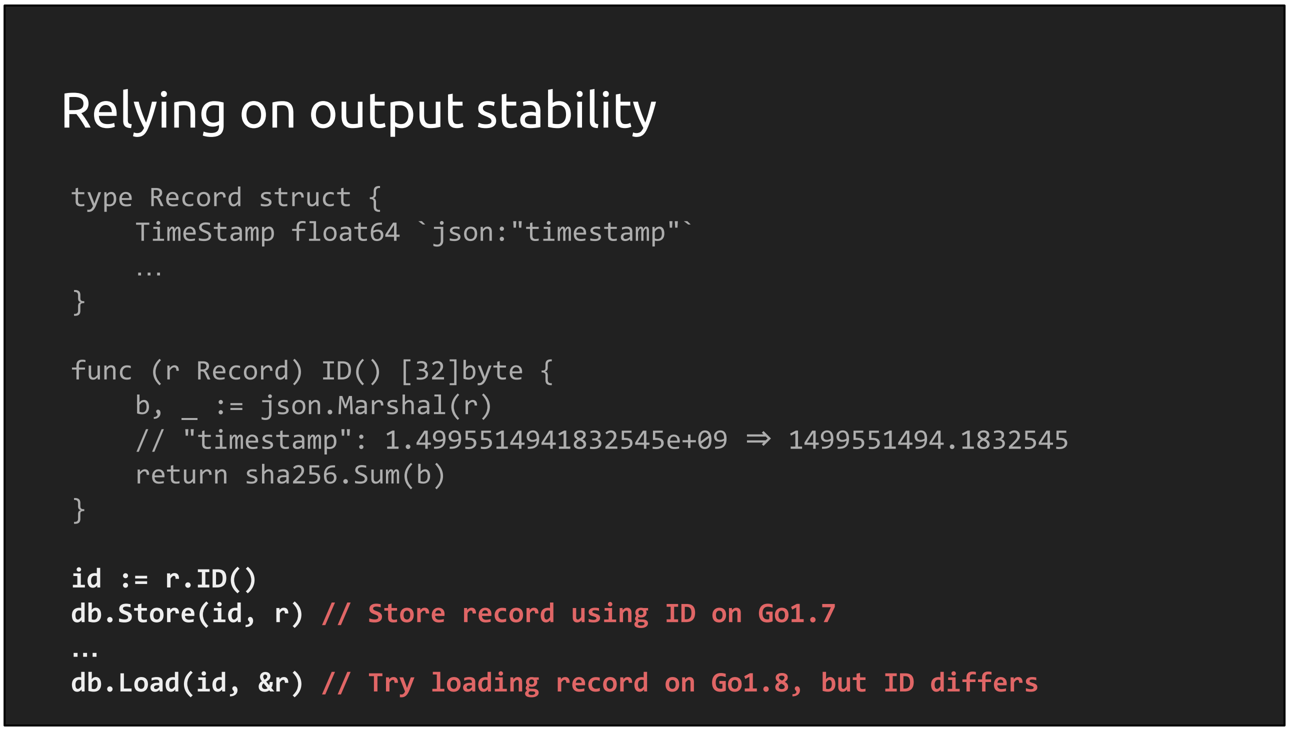
Relying on output stability

```
type Record struct {
    TimeStamp float64 `json:"timestamp"`
    …
}

func (r Record) ID() [32]byte {
    b, _ := json.Marshal(r)
    // "timestamp": 1.4995514941832545e+09 ⇒ 1499551494.1832545
    return sha256.Sum(b)
}

id := r.ID()
db.Store(id, r) // Store record using ID on Go1.7
…
db.Load(id, &r) // Try loading record on Go1.8, but ID differs
```

In order to obtain a unique identifier for each Record, an ID method returns the SHA-256 checksum of the JSON encoded struct.
However, as seen in blue, this is problematic because the JSON representation of the timestamp differs between releases.
On Go1.7, the output of serializing floats in JSON used exponential notation, while Go1.8 preferred decimal notation for a wider range of values.



Relying on output stability

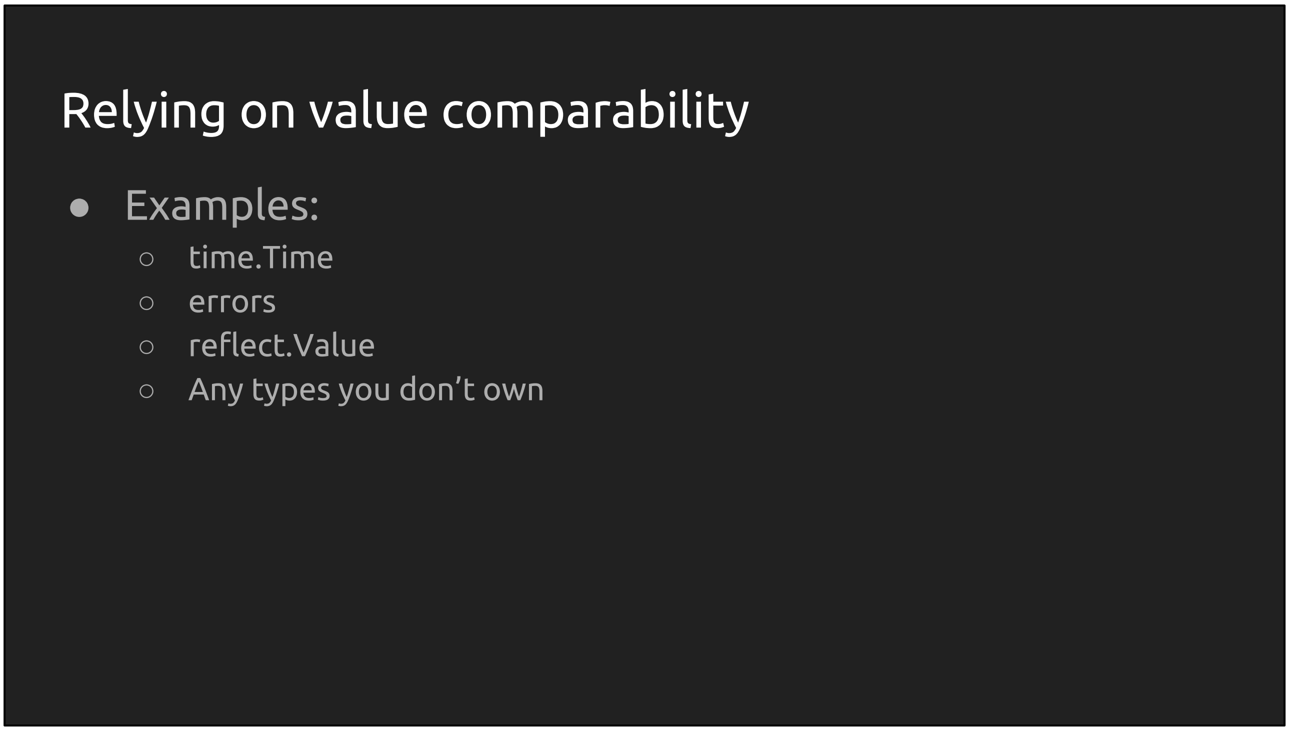
```
type Record struct {
    TimeStamp float64 `json:"timestamp"`
    …
}

func (r Record) ID() [32]byte {
    b, _ := json.Marshal(r)
    // "timestamp": 1.4995514941832545e+09 ⇒ 1499551494.1832545
    return sha256.Sum(b)
}

id := r.ID()
db.Store(id, r) // Store record using ID on Go1.7
…
db.Load(id, &r) // Try loading record on Go1.8, but ID differs
```

When this Record is stored into the database on one release of Go,
 it can not be properly retrieved in a later release since the ID has changed.
Bugs of this nature are surprising and difficult to track down.

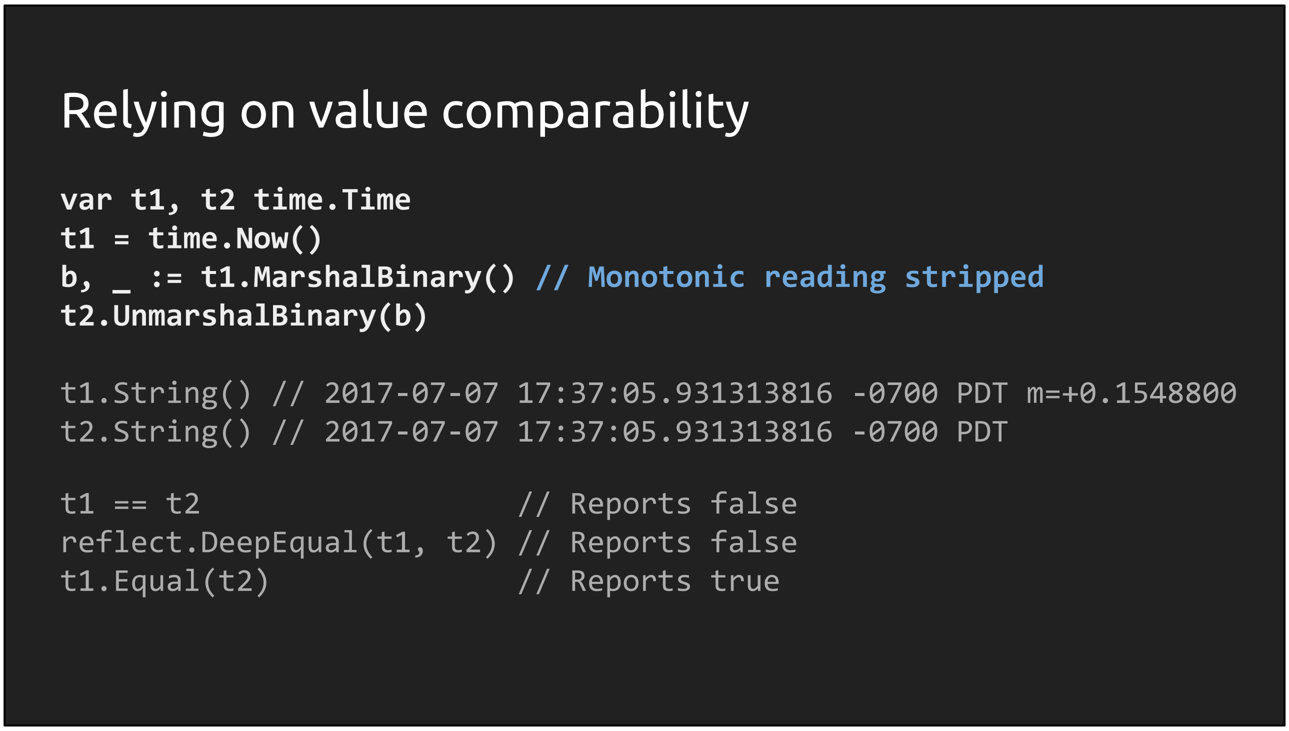
One way to fix this is to write your own struct marshaling
 where you can guarantee output stability or
 use a package that guaranteeds canonical serialization,
 which has the property of output stability.



Relying on value comparability

- Examples:
  - time.Time
  - errors
  - reflect.Value
  - Any types you don't own

The second category of erroneous assumptions is
assuming that all values may be directly compared using the equality operator.
Several types that are generally not safe to directly compare are
time.Time, errors, and any type you don't own (unless otherwise documented).



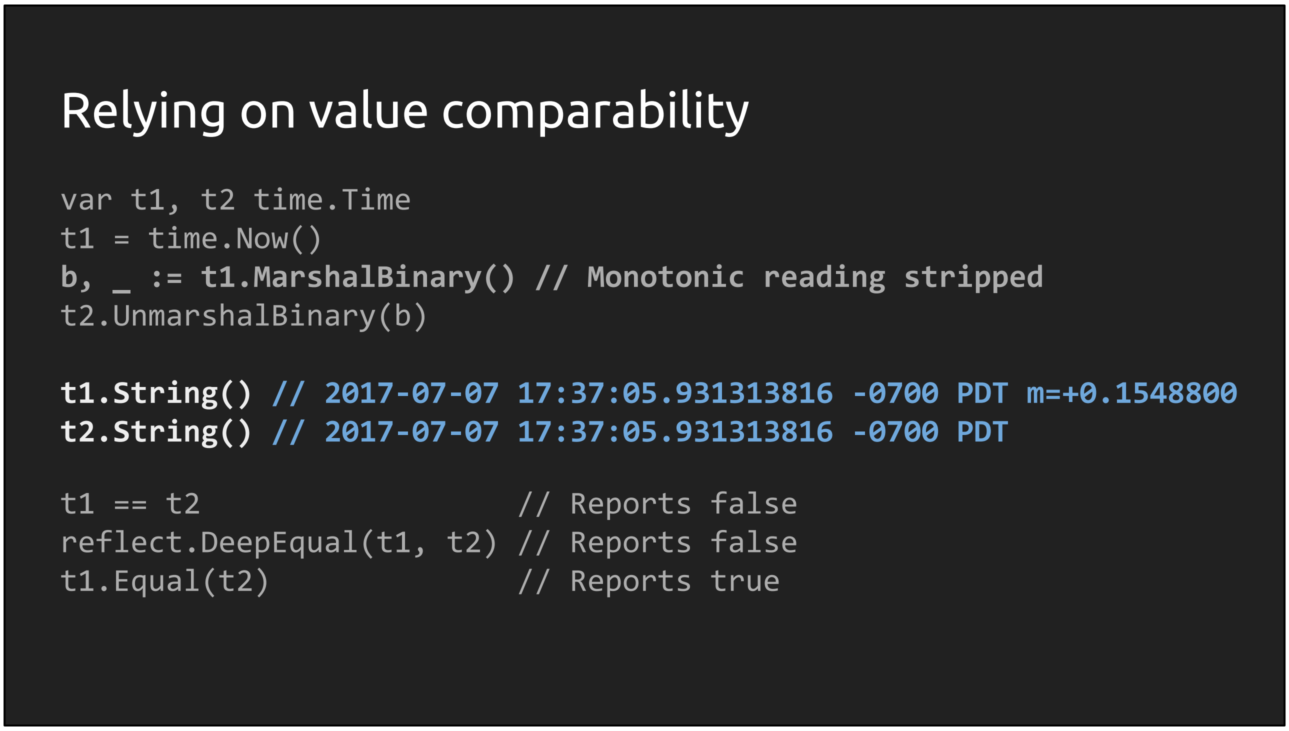
Relying on value comparability

```
var t1, t2 time.Time
t1 = time.Now()
b, _ := t1.MarshalBinary() // Monotonic reading stripped
t2.UnmarshalBinary(b)

t1.String() // 2017-07-07 17:37:05.931313816 -0700 PDT m=+0.1548800
t2.String() // 2017-07-07 17:37:05.931313816 -0700 PDT

t1 == t2                   // Reports false
reflect.DeepEqual(t1, t2) // Reports false
t1.Equal(t2)               // Reports true
```

Relevant to the upcoming Go1.9 release is the recording of a monotonic clock
into the Time type which provides more precise measurements of elapsed time.
By design, the monotonic reading cannot be serialized,
so round-trip marshaling and unmarshaling a Time loses any monotonic information.



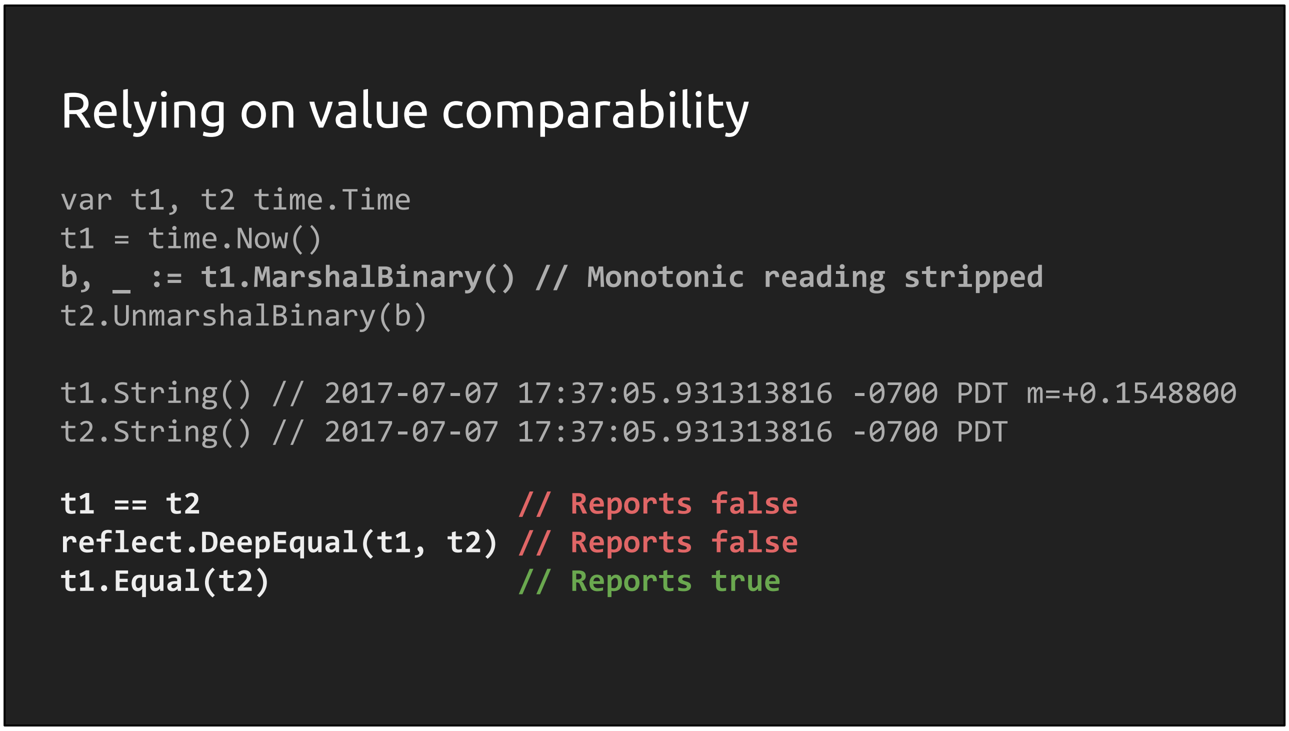
Relying on value comparability

```
var t1, t2 time.Time
t1 = time.Now()
b, _ := t1.MarshalBinary() // Monotonic reading stripped
t2.UnmarshalBinary(b)

t1.String() // 2017-07-07 17:37:05.931313816 -0700 PDT m=+0.1548800
t2.String() // 2017-07-07 17:37:05.931313816 -0700 PDT

t1 == t2                   // Reports false
reflect.DeepEqual(t1, t2) // Reports false
t1.Equal(t2)               // Reports true
```

As seen in blue, t1 contains a monotonic reading, while t2 does not.
Other than that, they represent the exact same time instant.



Relying on value comparability

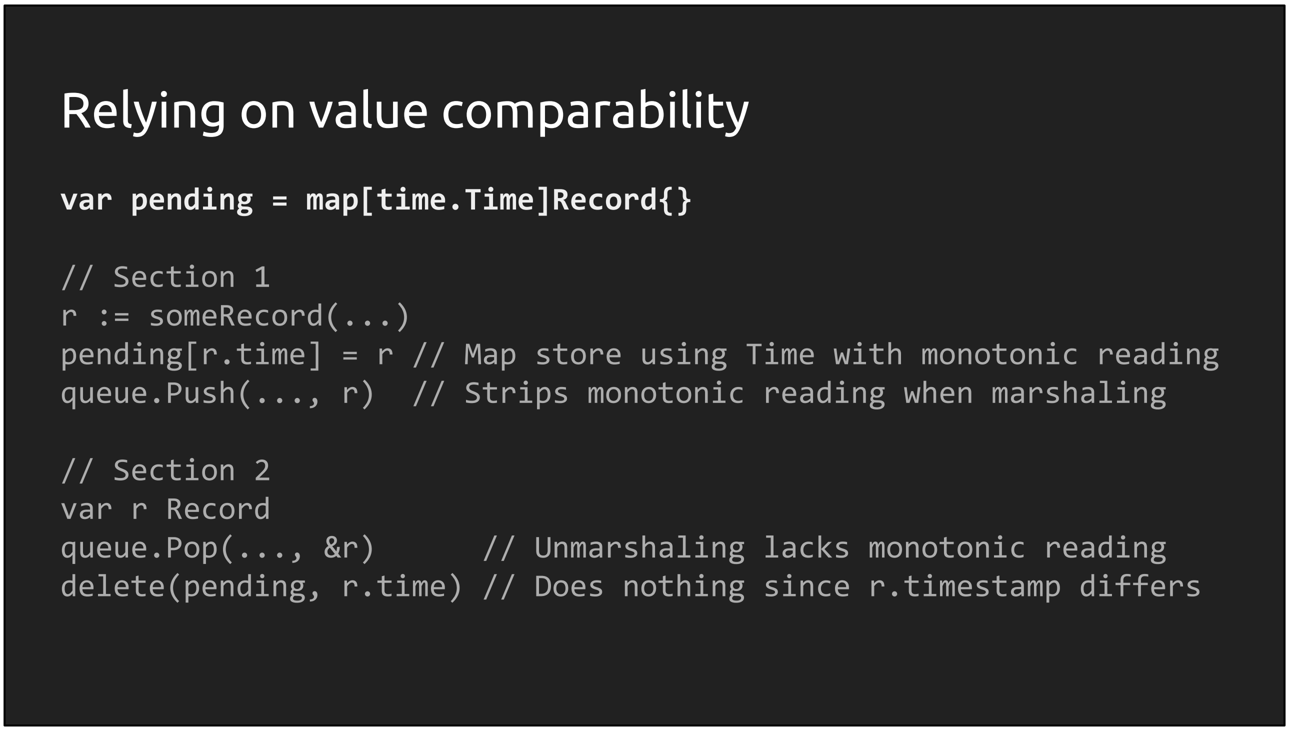
```go
var t1, t2 time.Time
t1 = time.Now()
b, _ := t1.MarshalBinary() // Monotonic reading stripped
t2.UnmarshalBinary(b)

t1.String() // 2017-07-07 17:37:05.931313816 -0700 PDT m=+0.1548800
t2.String() // 2017-07-07 17:37:05.931313816 -0700 PDT

t1 == t2                  // Reports false
reflect.DeepEqual(t1, t2) // Reports false
t1.Equal(t2)              // Reports true
```

At the bottom, we see that both the equality operator and reflect.DeepEqual
report that t1 and t2 are different.
However, the Equal method properly reports that these two times are the same.
If you read the documentation for Time, you will see that it
actually encourages use of the Equal method instead of the equality operator.
The reason why the equality operator gets this wrong is because it compares the underlying unexported fields of Time
without taking into account that the same time instant can have multiple representations.

Relying on comparability of Time can lead to serious bugs.



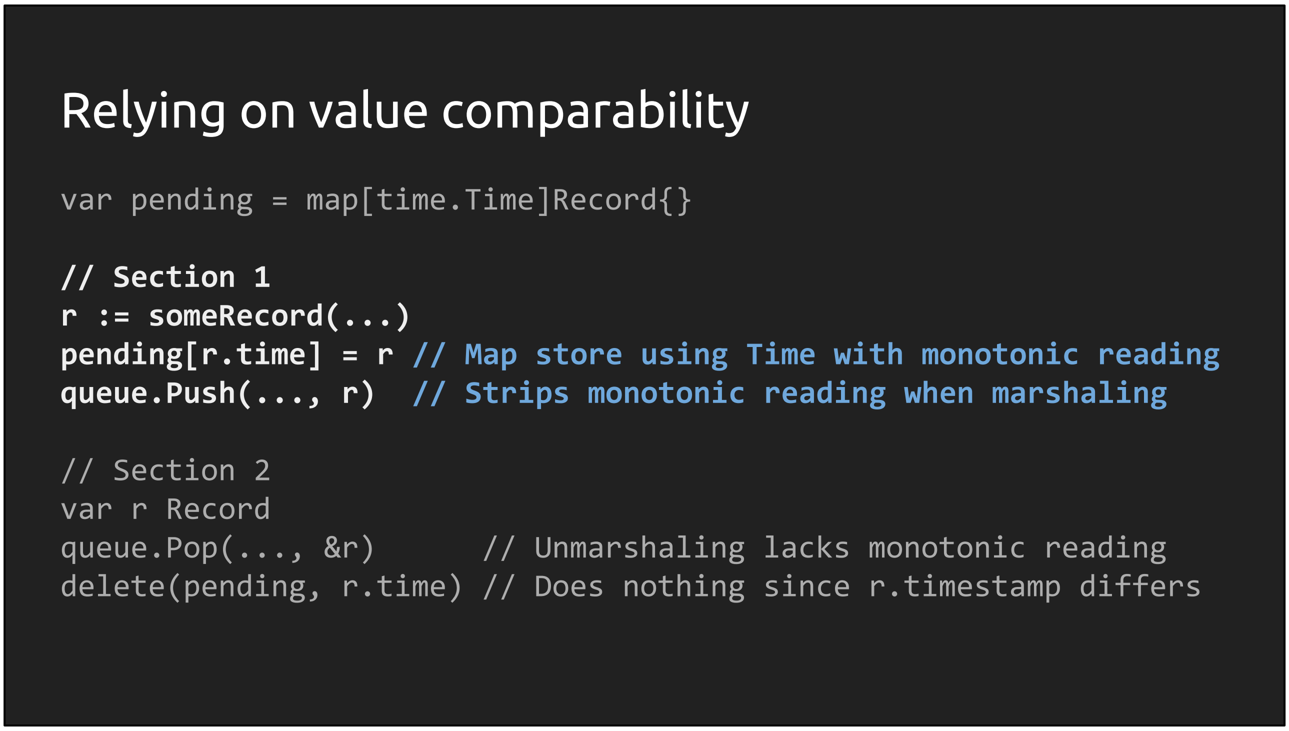
Relying on value comparability

```
var pending = map[time.Time]Record{}

// Section 1
r := someRecord(...)
pending[r.time] = r // Map store using Time with monotonic reading
queue.Push(..., r)  // Strips monotonic reading when marshaling

// Section 2
var r Record
queue.Pop(..., &r)      // Unmarshaling lacks monotonic reading
delete(pending, r.time) // Does nothing since r.timestamp differs
```

In this example, we have a shared map called pending
that is keyed by Time and
is accessed by two different sections of code.



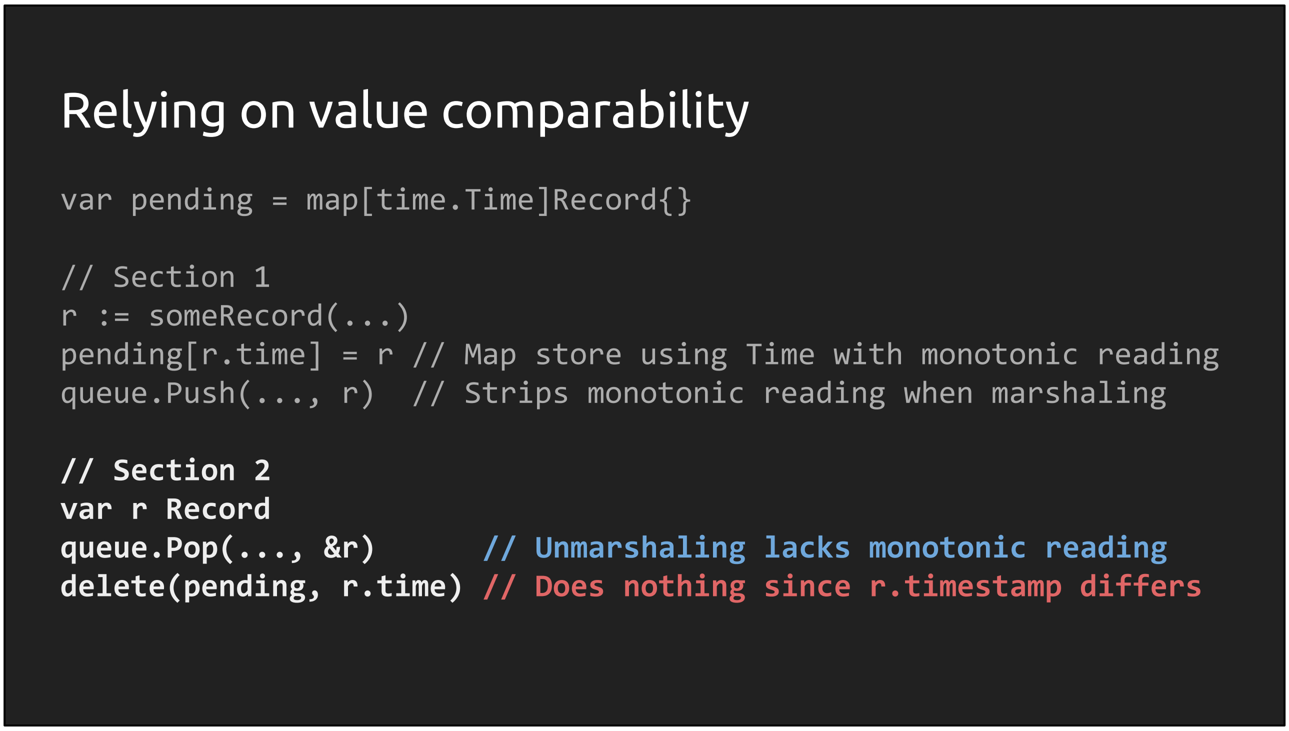
Relying on value comparability

```
var pending = map[time.Time]Record{}

// Section 1
r := someRecord(...)
pending[r.time] = r // Map store using Time with monotonic reading
queue.Push(..., r)  // Strips monotonic reading when marshaling

// Section 2
var r Record
queue.Pop(..., &r)      // Unmarshaling lacks monotonic reading
delete(pending, r.time) // Does nothing since r.timestamp differs
```

The first section inserts a record into the pending map using the timestamp as a key and
then stores that record into a persistent queue for processing.
The map insertion uses a time key with the monotonic reading and
the queue push strips the monotonic reading when marshaling.



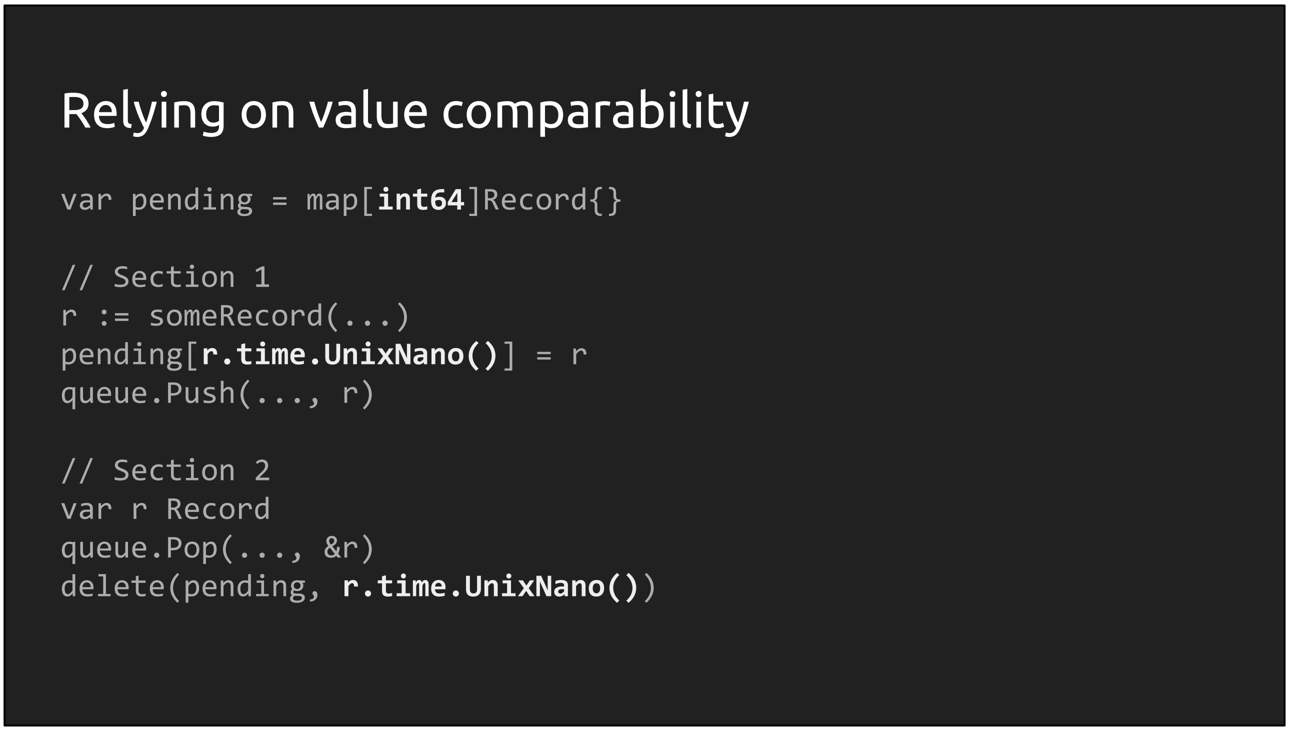
Relying on value comparability

```
var pending = map[time.Time]Record{}

// Section 1
r := someRecord(...)
pending[r.time] = r // Map store using Time with monotonic reading
queue.Push(..., r)  // Strips monotonic reading when marshaling

// Section 2
var r Record
queue.Pop(..., &r)       // Unmarshaling lacks monotonic reading
delete(pending, r.time) // Does nothing since r.timestamp differs
```

In a later section, the record is popped from the queue, and
an attempt is made to delete that record from the pending map.
The queue load and map deletion uses a time key that lacks a monotonic reading.
This is problematic and the deletion does not occur due to differing keys.



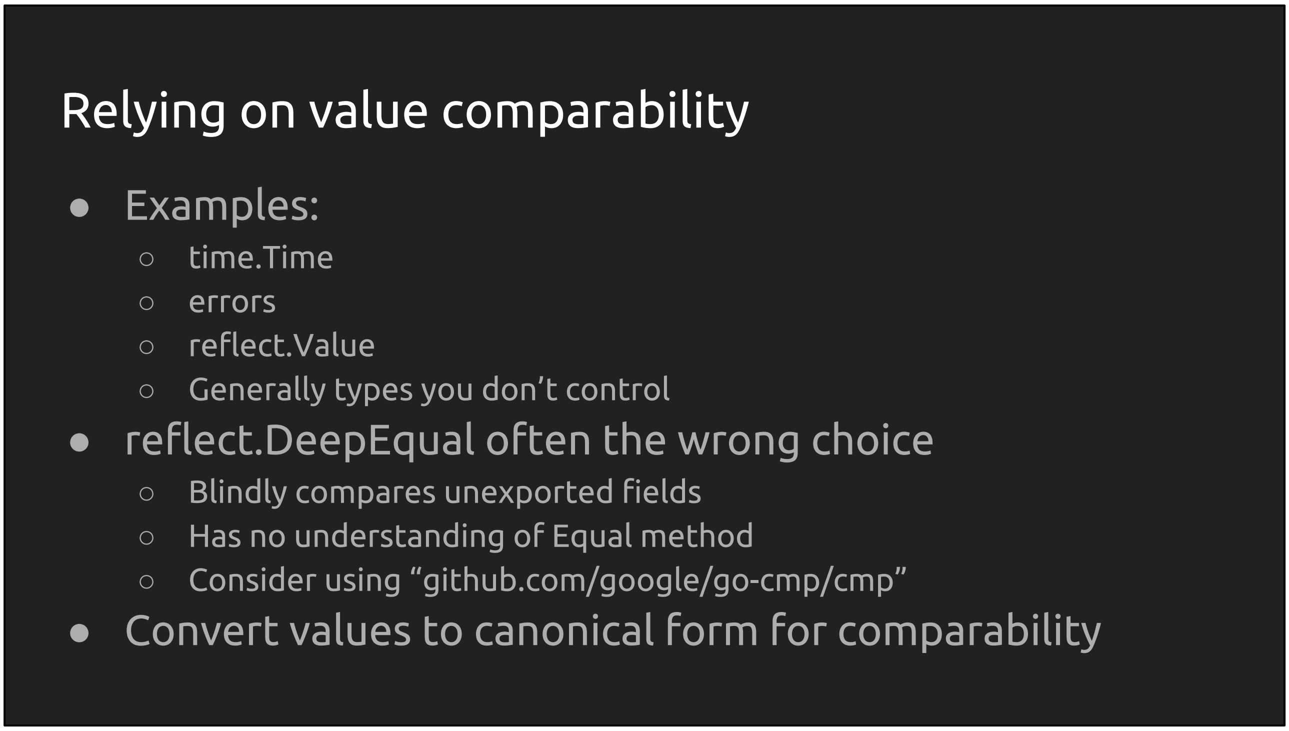
Relying on value comparability

```
var pending = map[int64]Record{}

// Section 1
r := someRecord(...)
pending[r.time.UnixNano()] = r
queue.Push(..., r)

// Section 2
var r Record
queue.Pop(..., &r)
delete(pending, r.time.UnixNano())
```

One way to fix this is to convert the time to
the number of nanoseconds since the Unix epoch and
use that instead as your map key.



Relying on value comparability

- Examples:
  - time.Time
  - errors
  - reflect.Value
  - Generally types you don't control
- reflect.DeepEqual often the wrong choice
  - Blindly compares unexported fields
  - Has no understanding of Equal method
  - Consider using "github.com/google/go-cmp/cmp"
- Convert values to canonical form for comparability

You have just seen two examples where relying on value comparability resulted in buggy code.
How can you prevent this?

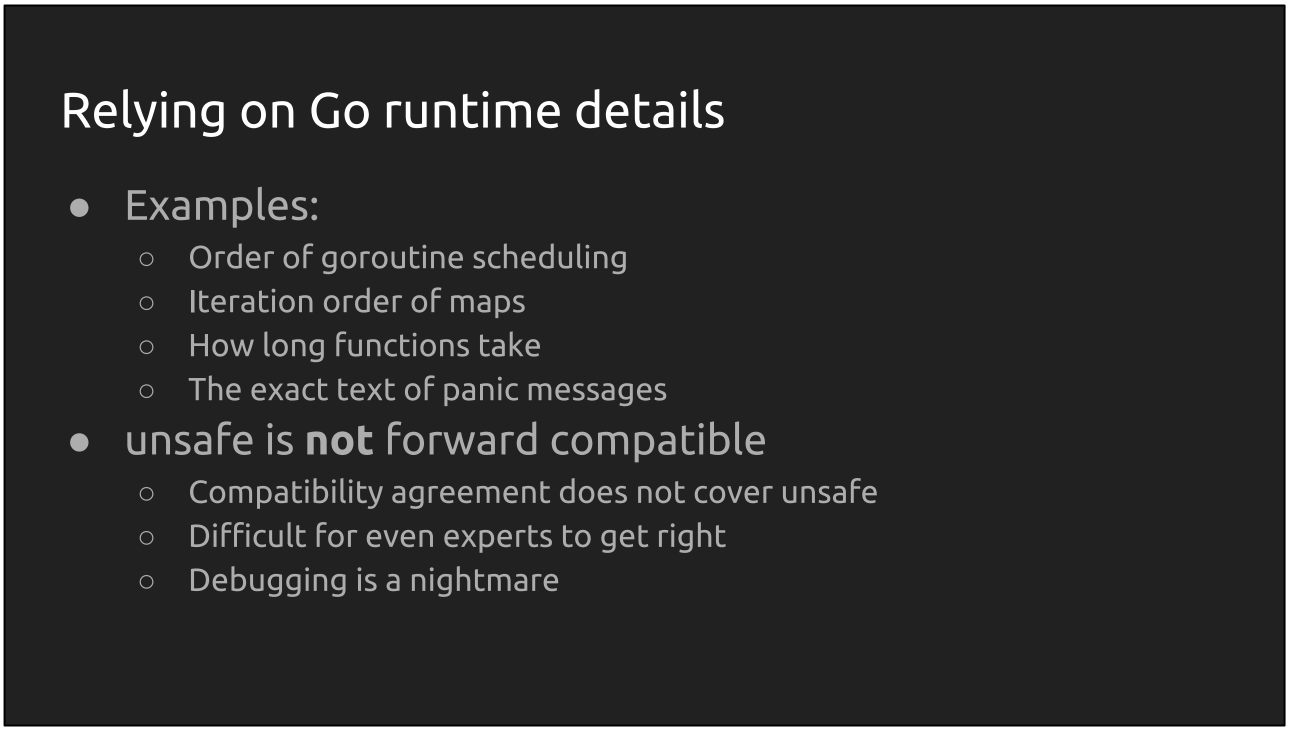
For tests, reflect.DeepEqual is often the wrong choice because it
    blindly compares on the internal details of types and
    has no understanding of Equal methods that package authors may have wanted you to use.
To deal with brittle tests inside Google,
    we have open-sourced a package on GitHub called cmp that avoids these two pitfalls.
We invite you to try using package cmp to improve your tests if it is the right fit.

Generally, in Go code, you should be careful about whether types are comparable and
    if they are not, you should either
    use custom definition of equality like the Equal method or
    convert the value to a canonical form that is stable.

Relying on Go runtime details

The third category of erroneous assumptions is relying on the internals of the Go runtime.

In each release of Go,
the compiler is producing faster code and
the garbage collector is getting smarter.
After release, it’s interesting to see the storm of bug reports
about how the latest release broke a user’s project,
when in fact, it only uncovered a pre-existing race condition.
The race detector has been instrumental in discovering these races
by intentionally randomizing aspects of the runtime
in an attempt to trigger detectable race conditions.

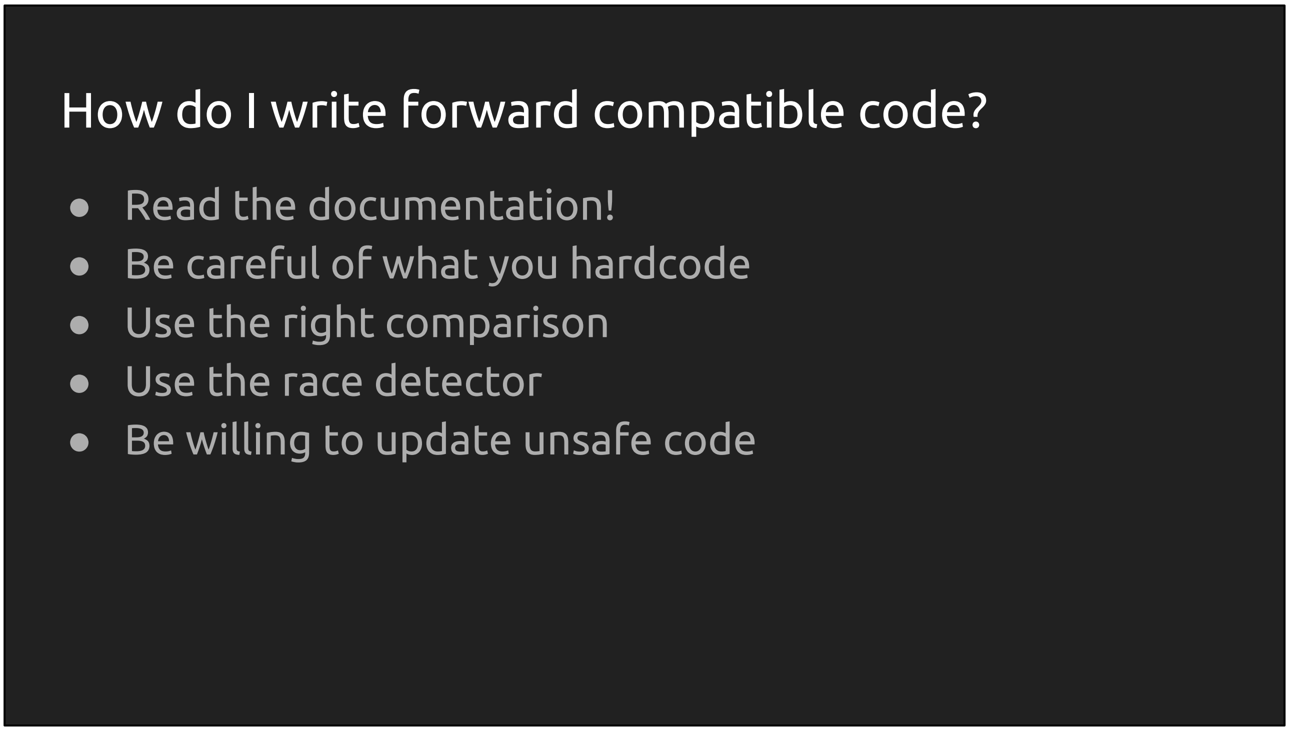


Relying on Go runtime details

- Examples:
  - Order of goroutine scheduling
  - Iteration order of maps
  - How long functions take
  - The exact text of panic messages
- unsafe is **not** forward compatible
  - Compatibility agreement does not cover unsafe
  - Difficult for even experts to get right
  - Debugging is a nightmare

An example of runtime details that users have relied upon to their detriment
is the ordering of when goroutines are scheduled,
which can be problematic when the ordering in which values appear in a channel
changes as they are produced by several goroutines.
Another example is relying on iterations over a map producing a deterministic ordering,
an assumption that was broken by an improved implementation of maps.
In order to reduce this type of failure,
since Go1.3, maps now provides a random iteration order.
Other examples include relying
on the exact timing of functions and
the exact text output of panic messages or stack traces.

One important detail about the runtime to mention is that
unsafe is not forward compatible.
The compatibility agreement actually reserves the right for the toolchain to make breaking changes.
We understand that use of unsafe is absolutely necessary in some applications,
so we try very hard to avoid breaking unsafe code unnecessarily, but
users of unsafe must be willing to update their code if necessary for future releases.
Build tags can be used to isolate behavior that exists in one release of Go and not another.



How do I write forward compatible code?

- Read the documentation!
- Be careful of what you hardcode
- Use the right comparison
- Use the race detector
- Be willing to update unsafe code

To summarize, how can you write forward compatible Go code?

First, I cannot stress this enough.
It is in your best interest to spend a few minutes reading the documentation carefully
in order to avoid introducing bugs
that will later take you hours to debug.
And if the documentation is not clear,
we would appreciate if you could file an issue or even contribute a fix!

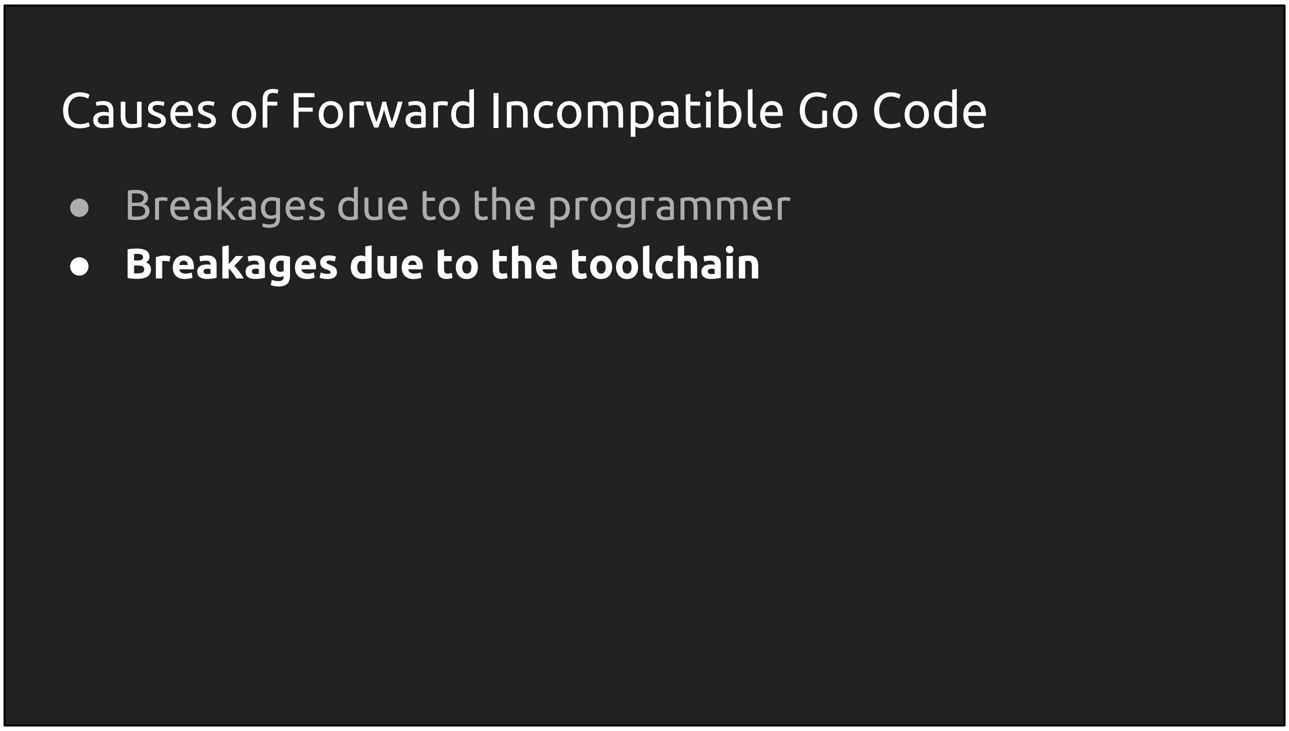
Secondly, be careful of what you hard-code.
Hard-coding values often accidently encodes assumptions about how
that data is generated or is to be used.
If these assumptions are not backed by guaranteed behavior in the documentation,
they may break your code in the future.

Third, use the right comparison.
In tests, you may consider using the cmp package that
we recently released instead of reflect.DeepEqual.
In production code,
you should either write your own equality functions or
use a canonicalized form of the value that is safe to compare.

Fourth, keep in mind that the race detector deliberately shakes up
runtime implementation details.

This can help you find existing race conditions that could manifest in future releases.

Lastly, if you use package unsafe, be willing to update your code before each toolchain release.

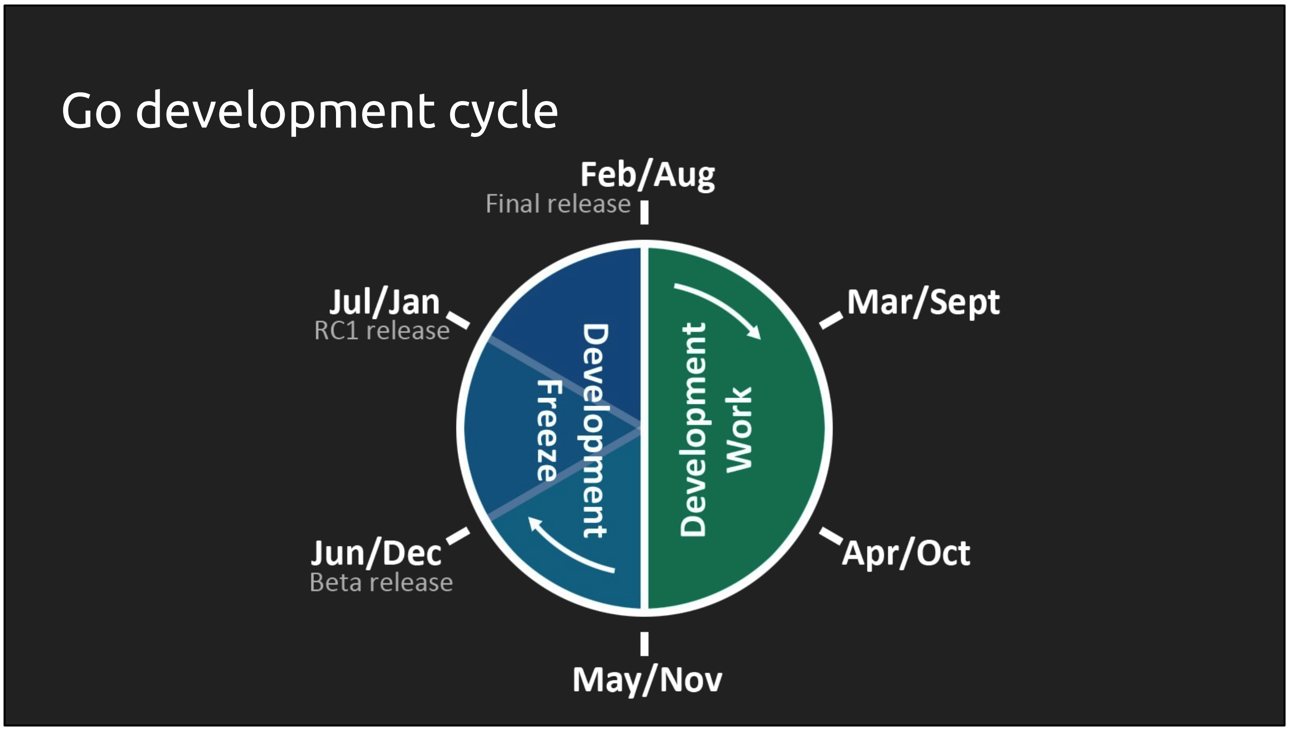


Causes of Forward Incompatible Go Code

- Breakages due to the programmer
- **Breakages due to the toolchain**

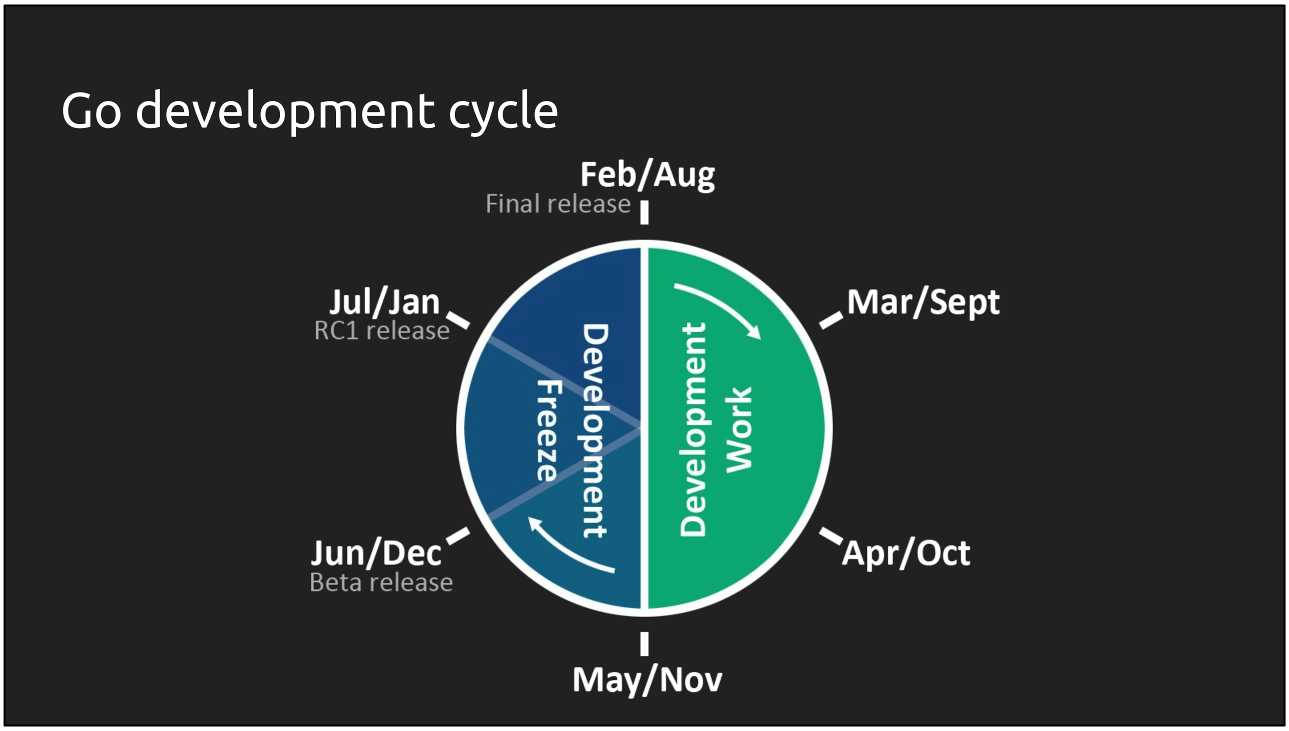
So we’ve looked at how breakages can occur because of
erroneous assumptions made by Go programmers.

The second cause of forward incompatible Go code is because
the toolchain changed some behavior that it promised to have.
In the development of the toolchain, we call these regression bugs, since
the behavior “regressed” relative to a prior release.
The developers of the Go toolchain are human just like you and
bugs can and do get introduced that can break your code;
And it’s entirely our fault.
As developers of the toolchain, we have the responsibility
that we adhere to the compatibility promise from our end.
In order to explain how we uphold the promise,
let’s talk about Go release process.



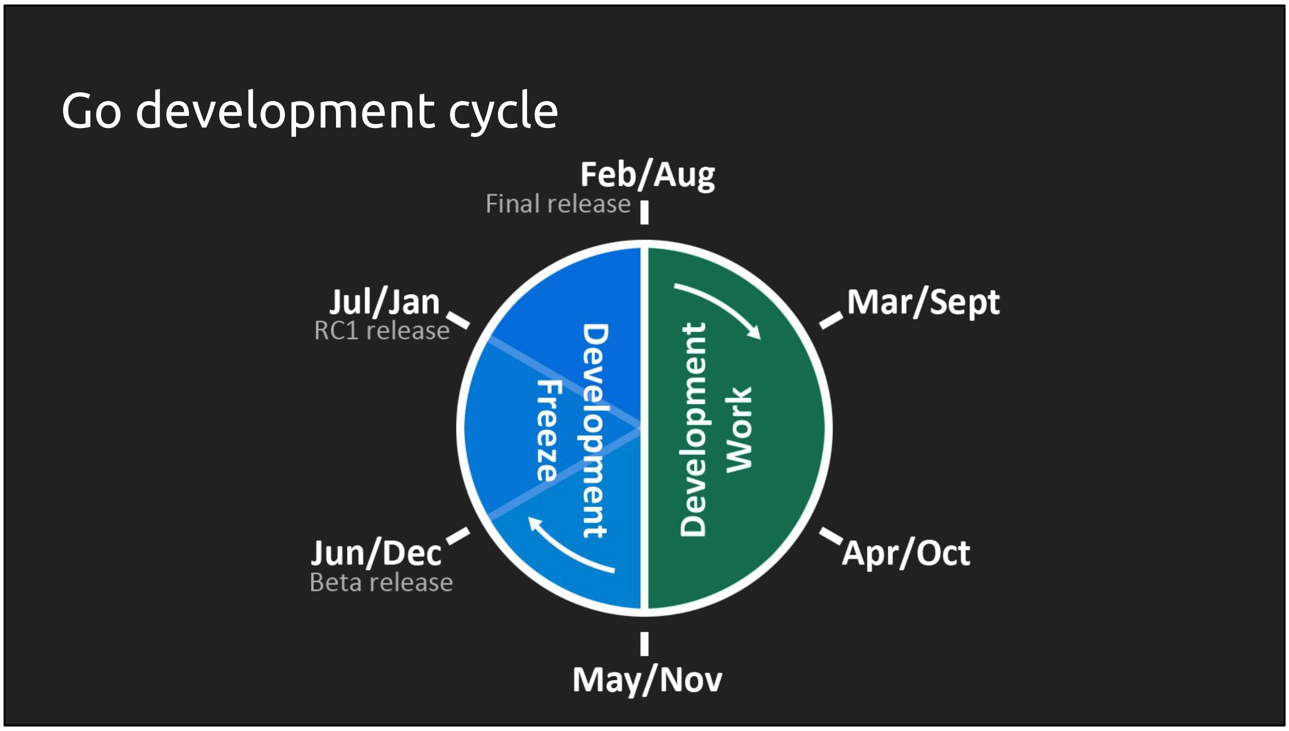
Go development cycle

The development of the Go toolchain is a 6-month process
that generally starts and ends in February and August.



Go development cycle

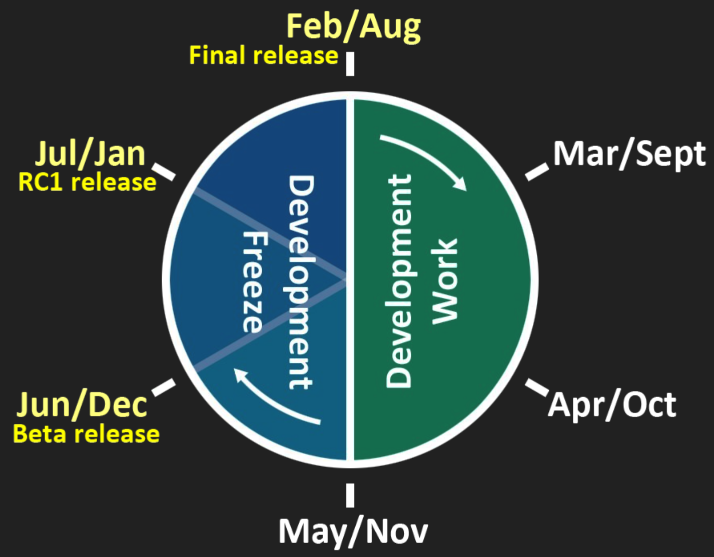
The first 3 months are spent
developing new features, optimizing performance, refactoring code,
and generally making higher risk changes.



Go development cycle

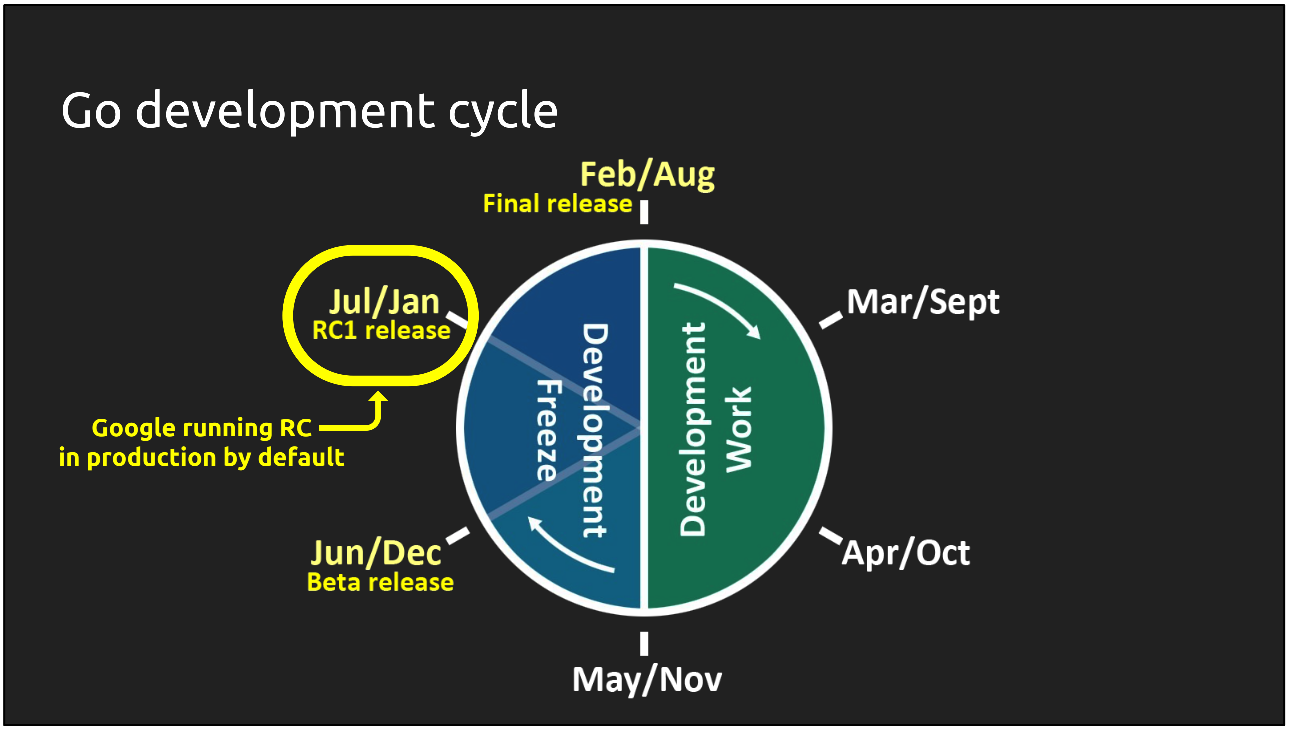
The later 3 months are spent in a development freeze, where we focus on
fixing bugs, improving documentation,
and generally stabilizing the toolchain.

Go development cycle



Throughout this process, we cut several preliminary snapshots of the toolchain
and invite the Go community to test these
in order to discover and fix bugs.
It is also an opportunity for the community to test-drive new APIs to discover
any unforseen pain-points that may result in adjustments to the API
before it becomes set in stone.

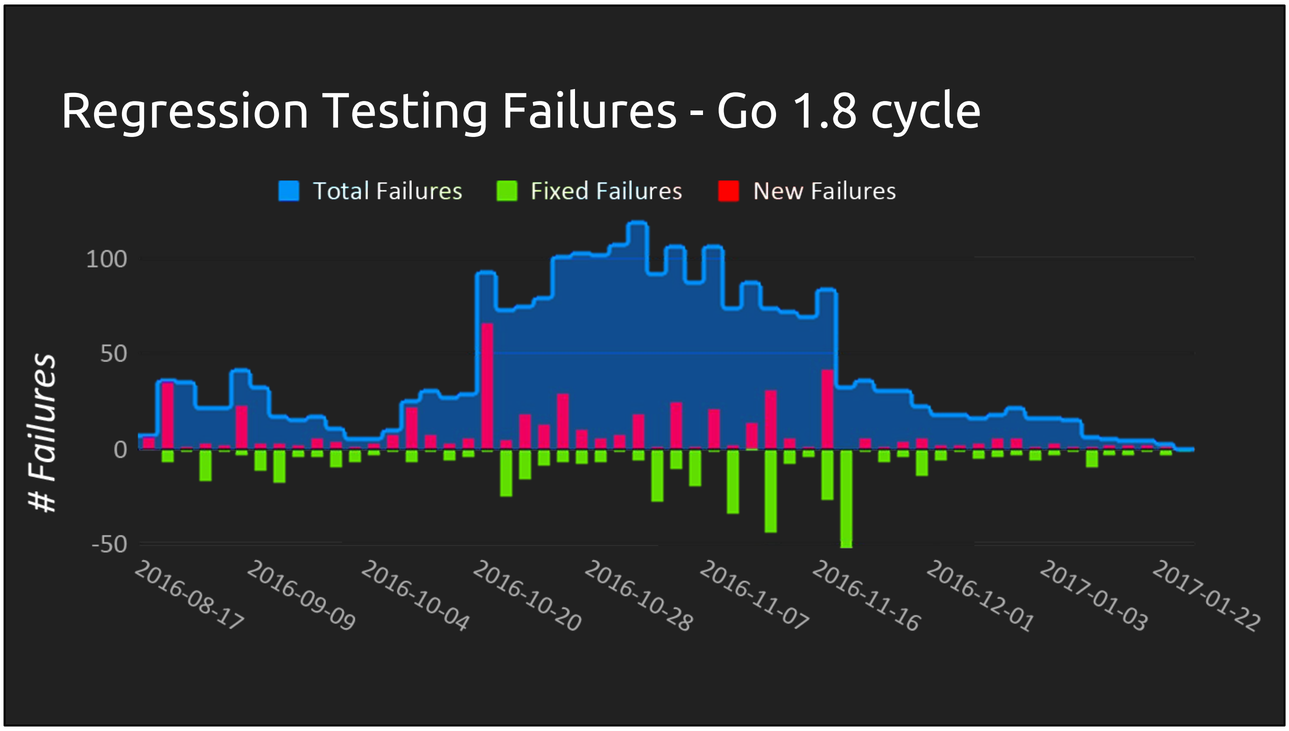
About 1 month into the freeze, the beta is cut and
is an indicative that most known bugs are fixed and
we are searching for unknown bugs.



Go development cycle

1 month later, the release-candidate is cut.
This is a significant milestone in the development cycle as this
cut of the toolchain is being used by default
for production builds inside Google.
All major bugs have been fixed and
it is a high-confidence statement by the Go team
that this cut is production quality.
As seen, the release-candidate is cut in July, and we hope to release it shortly.

After 1 month, at the end of the cycle after addressing new bug reports,
we have the final release,
which should only have minor changes from the release-candidate.

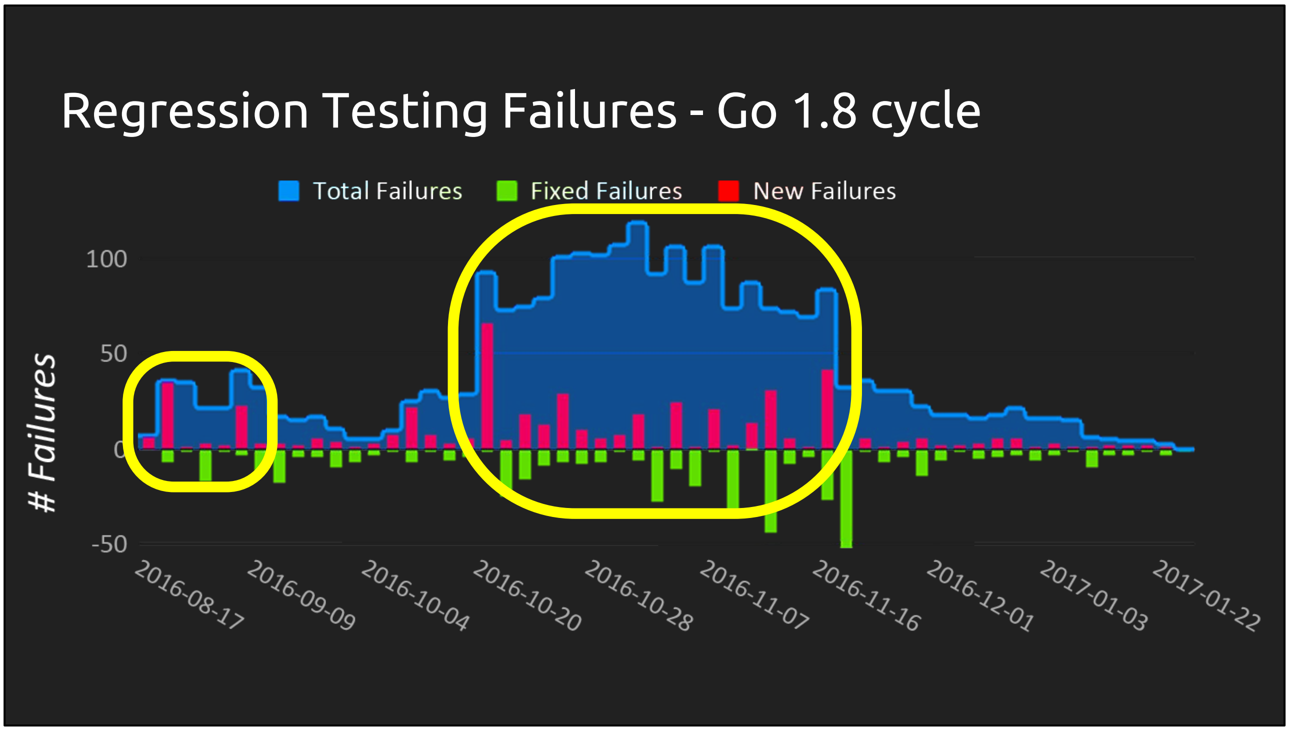


Throughout the entire development cycle, rigorous testing is performed.

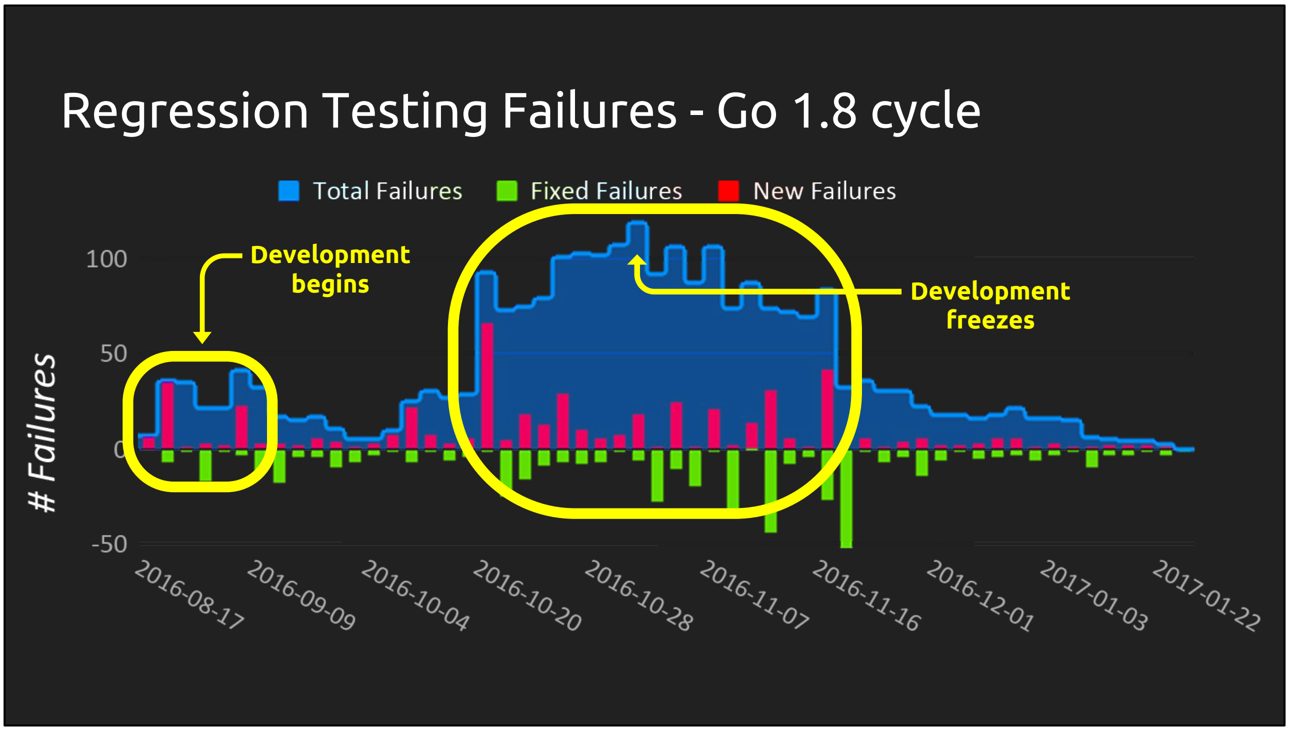
Every few days, a bleeding edge cut of the toolchain
is used to build, run, and test an enormous suite
consisting of over a million targets of Google's production code.
The purpose of this testing is to discover any regressions failures.
Some of these failures are the result of brittle code relying on erroneous assumptions,
others reveal a bug in the toolchain to fix upstream.

The chart above shows the number of regression failures
discovered on this test suite over the 1.8 cycle.
Each bar on the chart shows the results of a test run
where the blue bar tracks the current total number of failures,
the red bar tracks new failures relative to the previous run, and
the green bar tracks failures that have been fixed since the previous run.

Here are some observations.
First, even in a suite of a million targets,
we don't see more than several hundred failures,
which suggests either a fairly thorough code review process or
that we have a talented colony of contributing gophers.

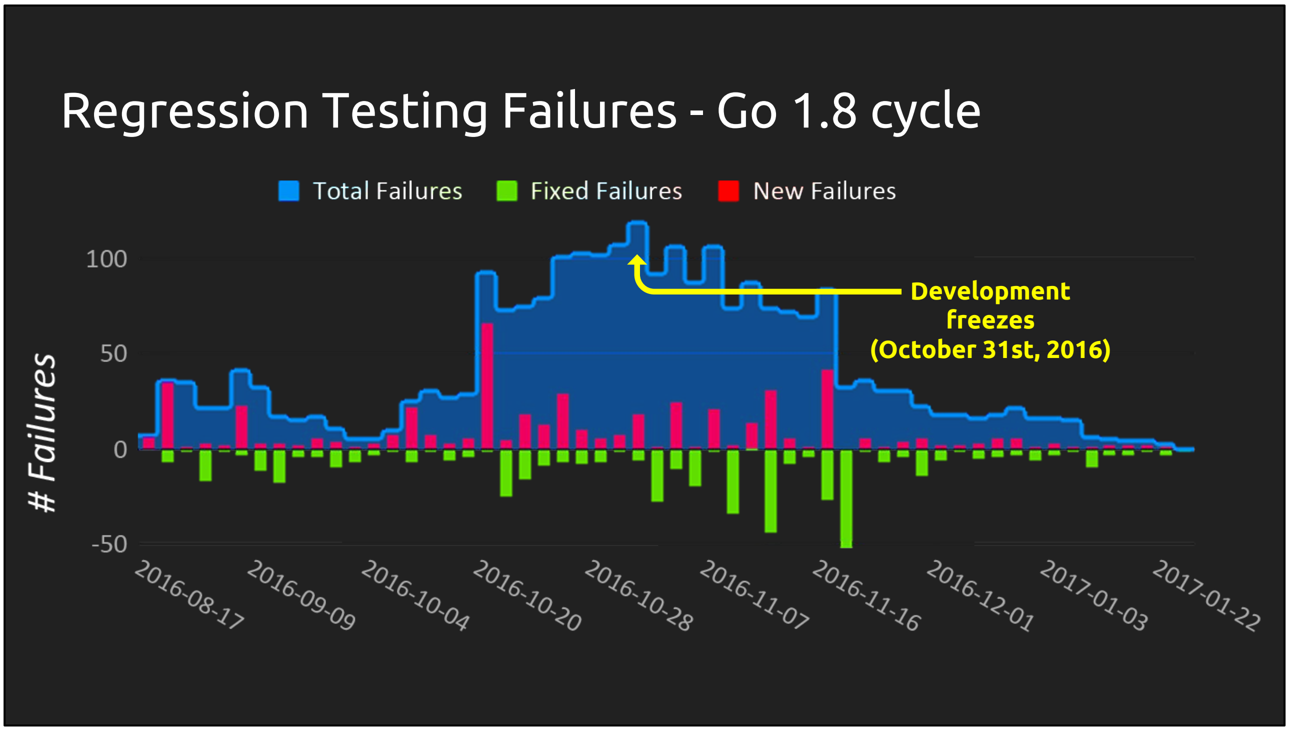


Second, in the cycle there two regions of high activity,
where a fair number of regression failures are introduced.

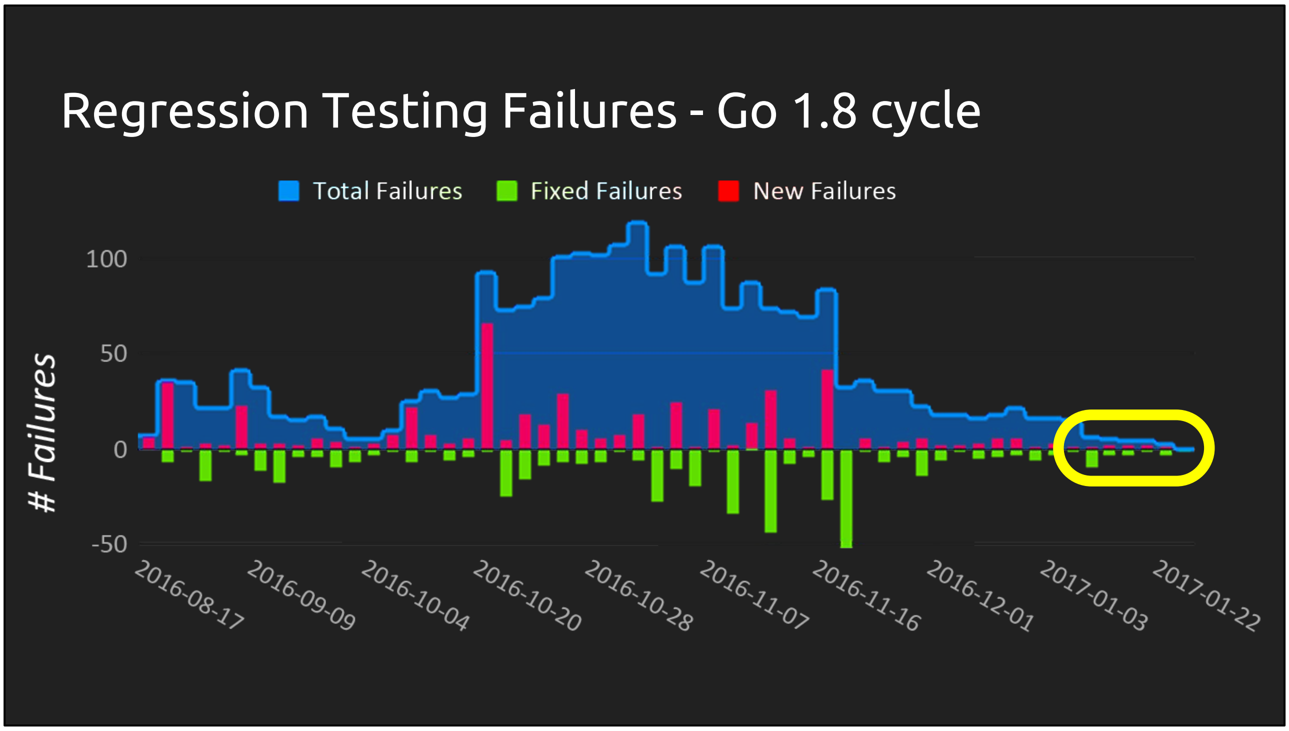


These regions correspond with the beginning of the development and the freeze of development.

You see, some gophers, love doing their work early,
batching up a number of changes to submit
the moment development opens.
Most other gophers (myself included), procrastinate until the development freeze,
frantically trying to push their changes through.
The number of regression failures is proportional to the number of feature changes submitted.

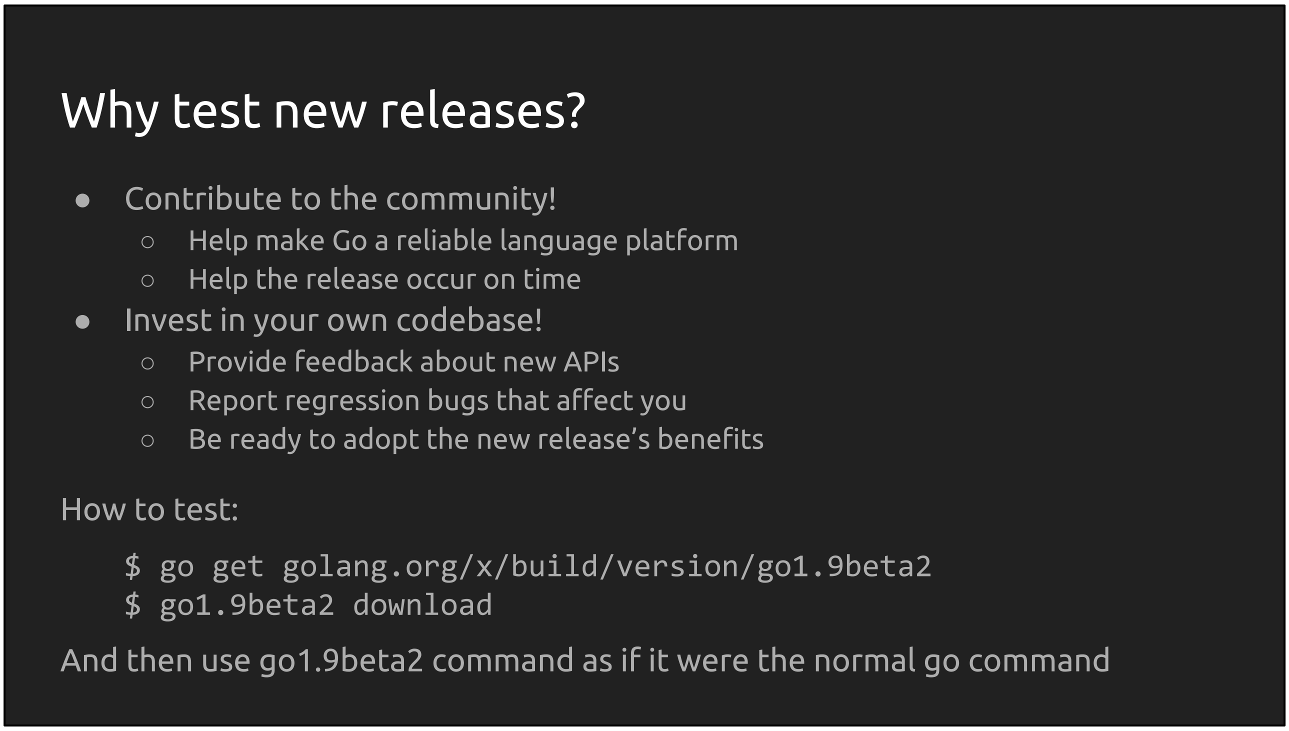


It is interesting to note that the climax occurs on October 31st, the day of the freeze.
See, the freeze is necessary as it intentionally impedes feature development
so that we can fix regression bugs those changes may introduce.
Without the freeze, these failures will continue to increase unbounded.



Lastly, as we approach the cycle's end,
once we fix all bugs that prevent Go from being production ready,
we have the release.

I share about our regression testing process to show
the Go team's commitment to make each Go release the best release.
However, as large as our test suite is,
it is still a tiny fraction of all the Go code in the world
and is heavily tailored towards Google's Go code.
Any codebase of sufficient size will use Go in a way
that is unique, and that we did not test for.
For that reason, we need your help!



Why test new releases?

- Contribute to the community!
  - Help make Go a reliable language platform
  - Help the release occur on time
- Invest in your own codebase!
  - Provide feedback about new APIs
  - Report regression bugs that affect you
  - Be ready to adopt the new release's benefits

How to test:

```
$ go get golang.org/x/build/version/go1.9beta2
$ go1.9beta2 download
```

And then use go1.9beta2 command as if it were the normal go command

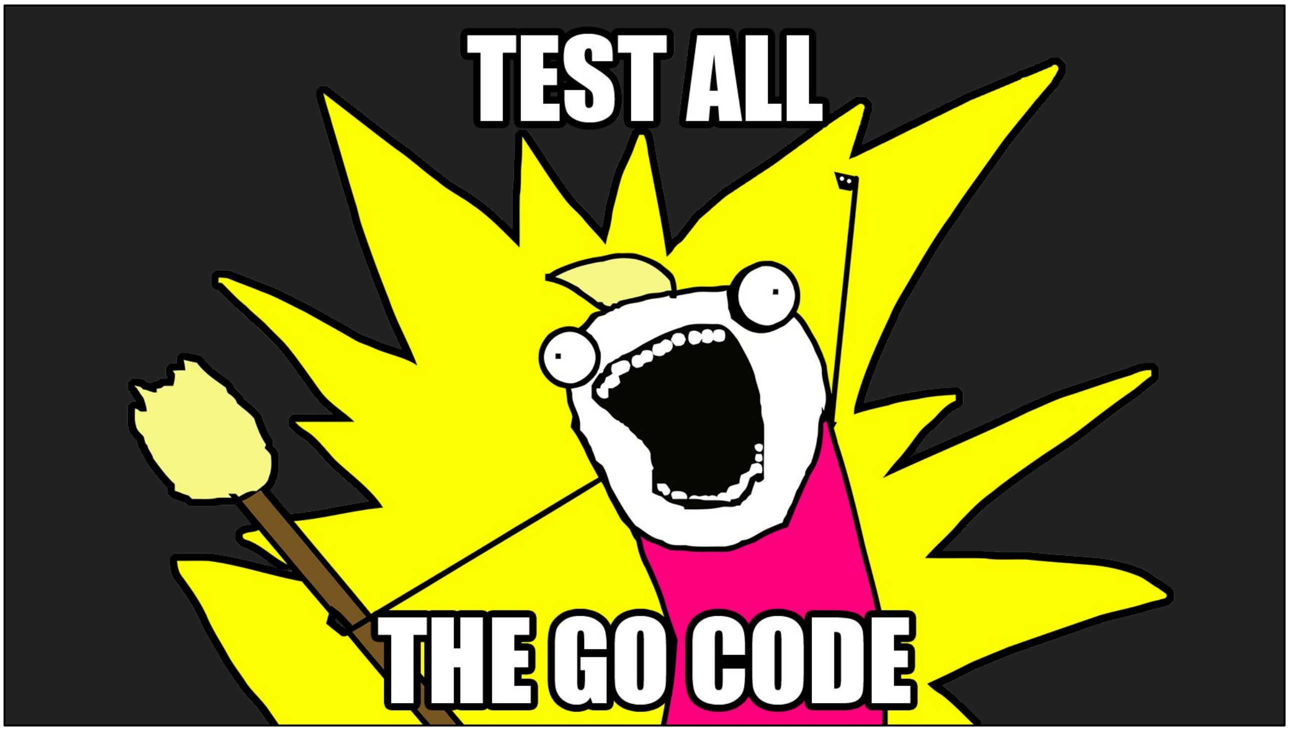
We need your help to test new releases of Go.
Here are some good reasons why.

First, it is an excellent way to contribute to Go community.
By reporting regression failures,
    you can help Go become the most reliable language platform;
    one that users have confidence in and choose over alternatives.
Also, reporting regressions early allows Go to be released on time,
    so that all of us may benefit sooner from upcoming features.

Second, testing early is an investment in your own codebase.
Open-source development of Go means
    you have a unique opportunity to try out new APIs.
If the API is clunky to use for your application,
    then early in the cycle is *the* time to provide that feedback,
    otherwise it's too late.
It's also important for you to report unique regression bugs that affect your code.
If you don't report them, we can't fix them.
Also, by shepherding your codebase and fixing brittle code,
    you set yourself up to benefit from exciting new features sooner.
(e.g., type aliases, monotonic timestamps, sync.Map, helper test funtions, or the bits package).

So, how can you easily test 1.9 today?
Actually, quite simply.
You can “go get” this special package: “go1.9beta2”,
then run “go1.9beta2 download” to install the new toolchain,
and then just invoke go1.9beta2 as if it were the normal go command.
Using this, you can run all your tests and
build a 1.9 binary for your production canaries.
If failures arise, triage them, and determine whether it’s because
of brittle code based on erroneous assumptions or
a regression bug in the toolchain.
If the later, we would appreciate
if you filed bug report with a succinct reproduction case,
so that we may address it.

Alright, so.... If I did my job correctly, I shouldn’t have to ask, but...



Who’s going to test Go 1.9!?

Great! I expect to see bug reports by the end of tonight!

Thank you for your time.