TAKUMA SHIBUYA / @sivchari

Go Conference Organizer • Kubernetes Maintainer

YOUR TESTS PASS, BUT DO THEY WORK?

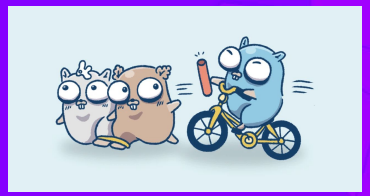
Mutation Testing in Go with gomu



ABOUT ME

- Takuma Shibuya (@sivchari)
- LayerX, Inc Platform Engineer
- Go Conference main organizer
- Kubernetes maintainer • Argo CD maintainer
- Kubernetes 2025 Contributor Award
- Author of sivchari/gomu







AGENDA

01 Why we write tests

02 What coverage misses

03 Mutation testing basics

04 Inside gomu: making mutants

05 Inside gomu: running mutants fast & safe

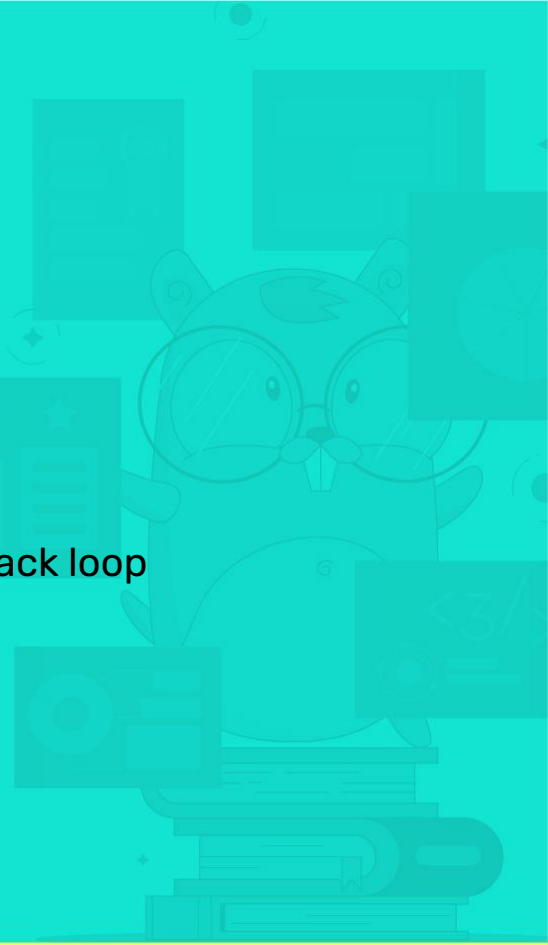
06 gomu in CI

01

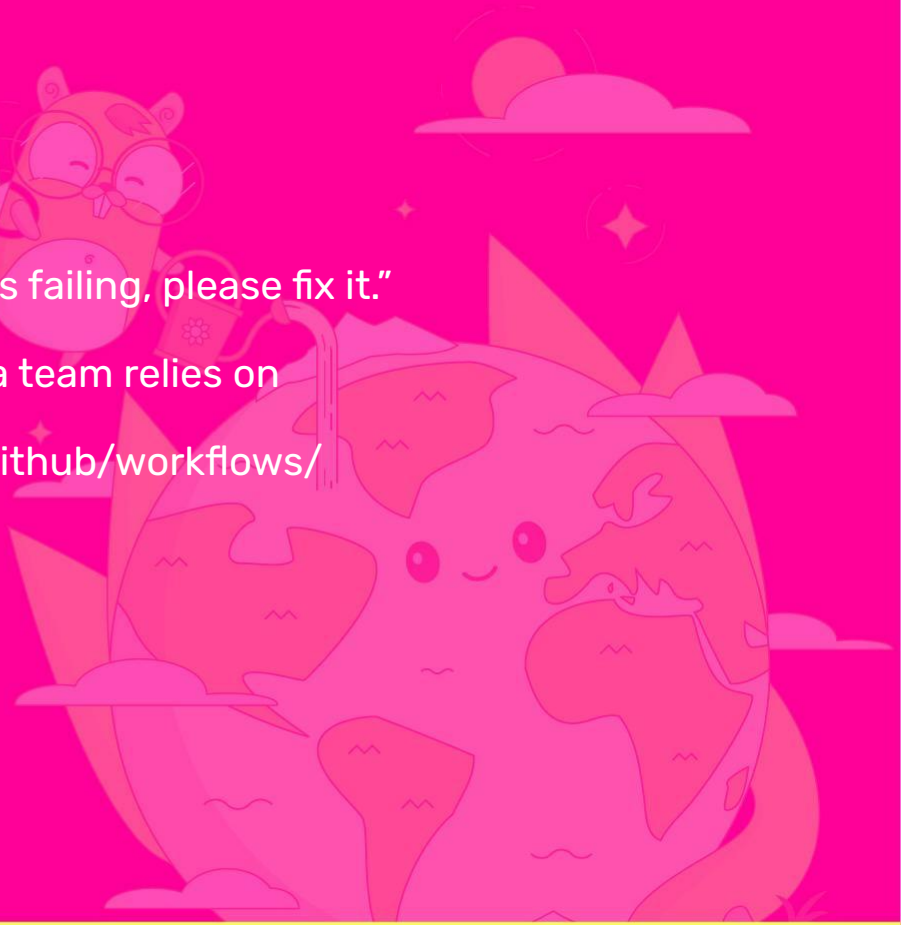
WHY WE WRITE TESTS

CI is the minimum agreement in team development, and now for AI agents too.

- CI as a shared agreement
- CI as the AI agent's feedback loop
- Tests as specification



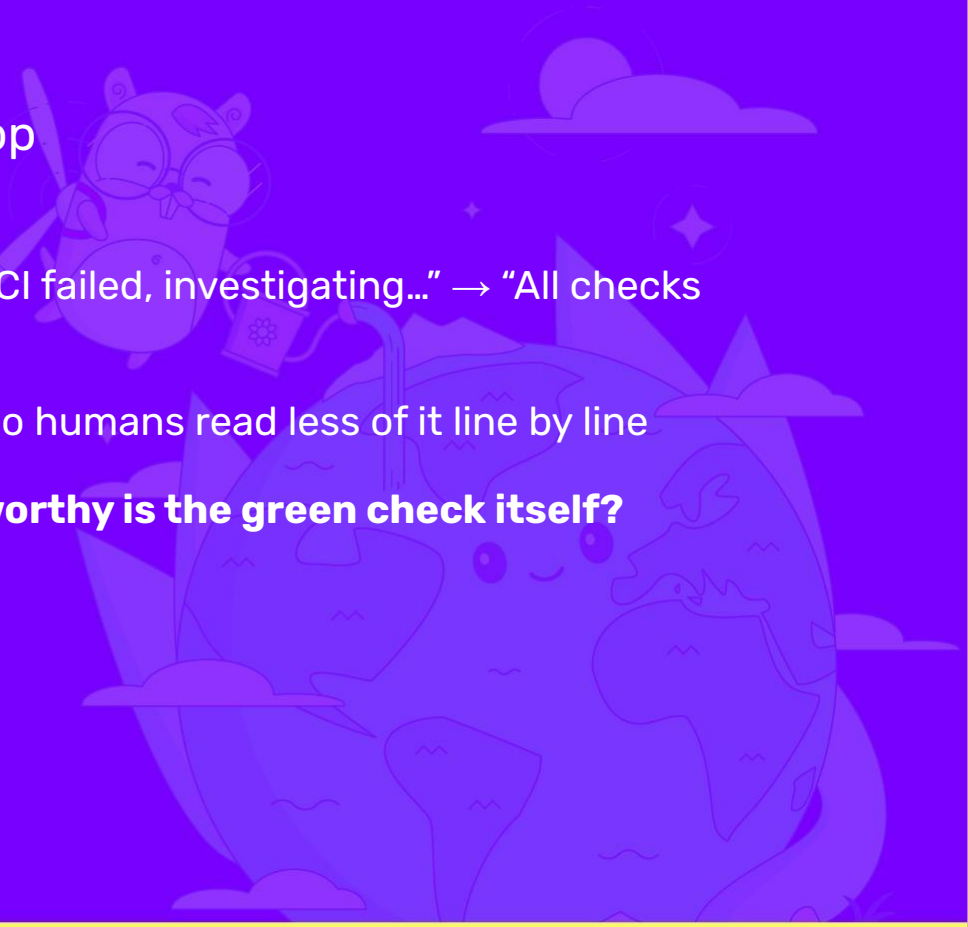
CI: the minimum agreement



- “CI passed, please review my PR.” / “CI is failing, please fix it.”
- A green check is the shared agreement a team relies on
- New project? The first commit is often .github/workflows/

CI is now the AI agent's feedback loop

- Agents judge their own work by CI: “CI failed, investigating…” → “All checks passed, done!”
- AI-generated code grows very fast, so humans read less of it line by line
- **So the real question is: how trustworthy is the green check itself?**



Go's tools

Static analysis

go vet · go fix

Dynamic verification

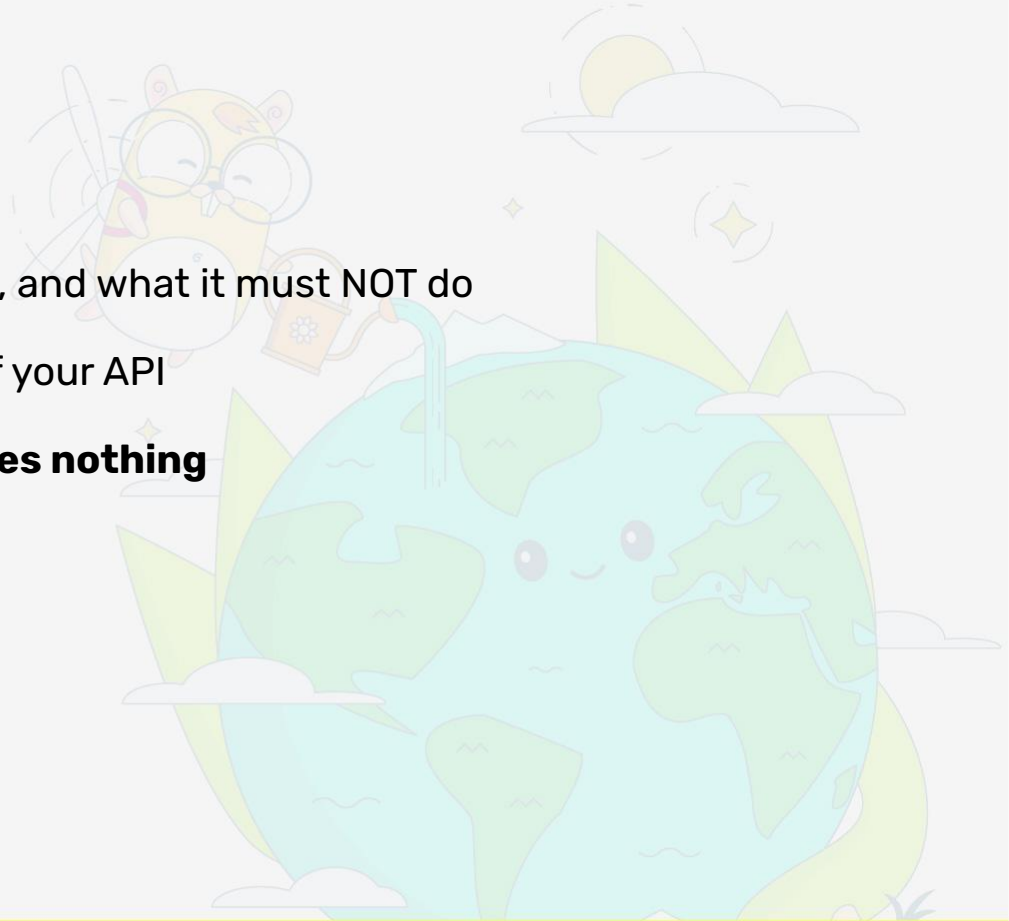
go test: unit, benchmark, example, fuzzing

All of it fits in a single CI job.

```yaml
name: Go
on: [push]
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v5
      - uses: actions/setup-go@v5
        with:
          go-version: '1.27'
      - run: go vet ./...
      - run: go fix -diff ./...
      - run: go test ./...
```

Tests are specifications

- White-box: what the code must do, and what it must NOT do
- Black-box: a test is the first user of your API
- **A test with no assertions specifies nothing**
- …but it still turns CI green.



02

WHAT COVERAGE MISSES

Coverage tells you what ran, not what was checked.

- How Go measures coverage
- Goodhart's law
- 100% coverage, zero verification

Coverage in Go

Statement coverage for unit tests.

Since Go 1.20, binaries built with -cover also produce coverage data, which is useful for integration and e2e tests.

Both answer: “was this line executed?”

```
$ go test -cover ./...
ok  example.com/calc  coverage: 100.0%

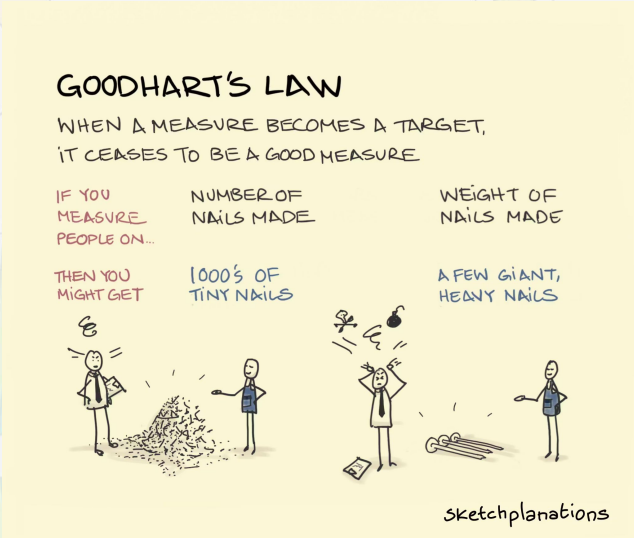
$ go test -coverprofile=c.out ./...
$ go tool cover -html=c.out

$ go build -cover -o app .
$ GOCOVERDIR=covdata ./app
$ go tool covdata percent -i=covdata
```

Goodhart's law

"When a measure becomes a target, it ceases to be a good measure."

- Typing speed → practicing only typing
- PR count → splitting PRs to raise the count
- 80% coverage → tests that only exist to run lines



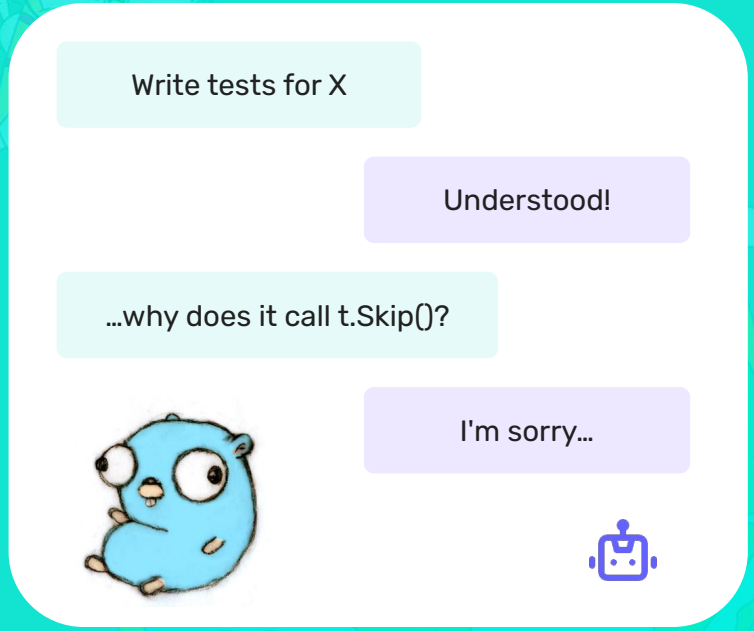
100% coverage, zero assertions

```go
func Div(a, b int) (int, error) {
    if b == 0 {
        return 0, errors.New("b must not be zero")
    }
    return a / b, nil
}

func TestDiv(t *testing.T) {
    _, _ = Div(1, 1) // every statement runs...
    _, _ = Div(1, 0) // ...nothing is checked
}
// coverage: 100.0% of statements
```

The AI era makes fake tests easier

- Agents optimize for the signal they can see: green CI
- t.Skip: go test still prints ok, exit 0. Coverage does drop, but nobody is watching it
- A test that calls your code and checks nothing: green CI and 100% coverage
- **We need a metric that fails when tests don't verify**



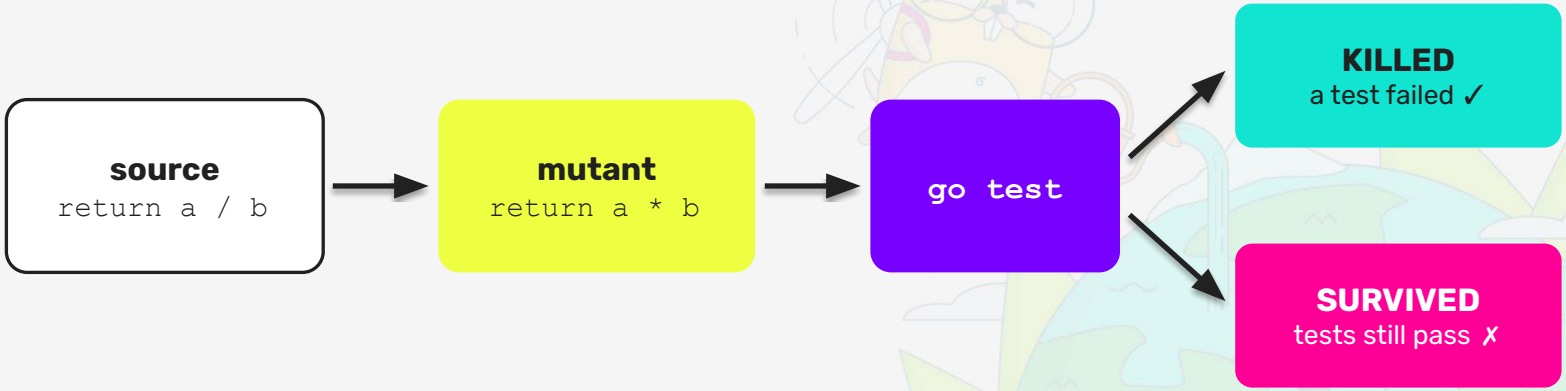
03

MUTATION TESTING

Test your tests by breaking your code on purpose.

- Mutants: KILLED & SURVIVED
- Div, revisited
- Costs & problems

How mutation testing works



Each mutant = one small, deliberate bug (a mutation operator applied to one AST node).
If no test fails on the bug, the behavior isn't really specified.

Mutant statuses

Mutation score

KILLED / (all − NOT_VIABLE)

SURVIVED mutants are a to-do list: each one is a behavior no test checks.

```
KILLED       a test failed
             -> good

SURVIVED     all tests passed
             -> not tested

NOT_VIABLE   mutant doesn't compile
             -> excluded from score

TIMED_OUT    reached --timeout
             -> e.g. infinite loop

ERROR        preparation failure
```

Div vs. the test with no assertions: real gomu output

```
$ gomu run .        calc.go (16 mutants) -> 5/16 killed

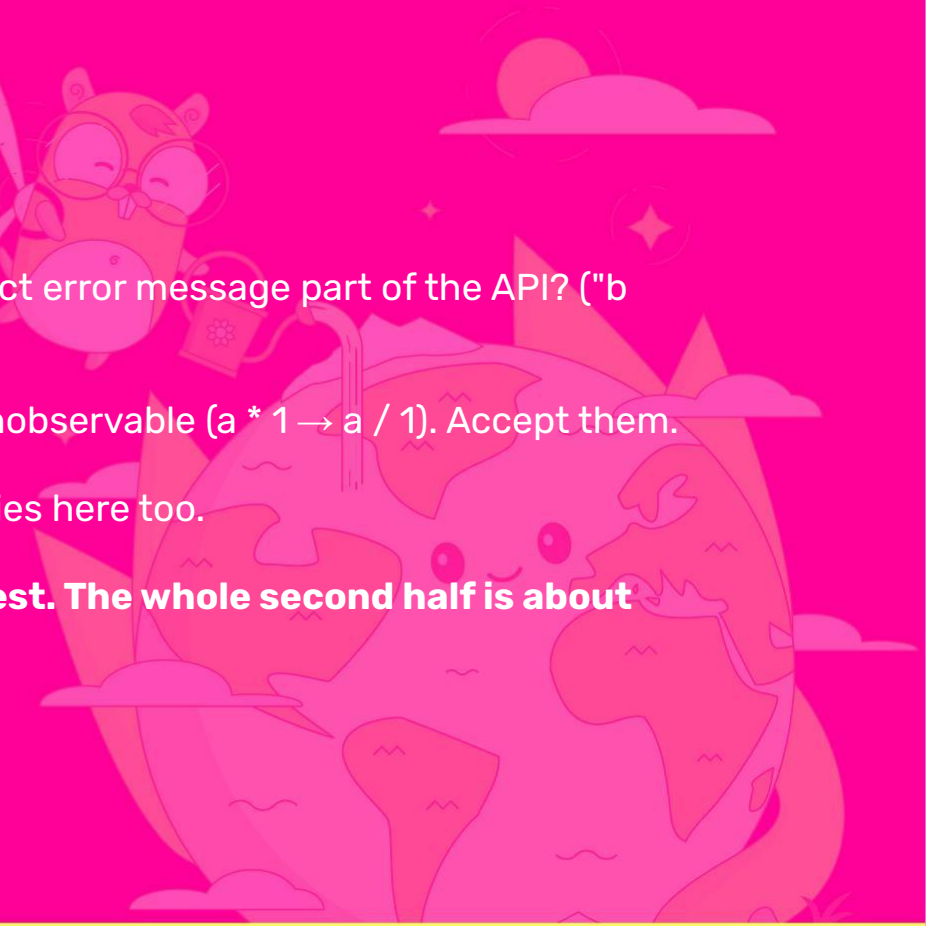
L6  branch              b == 0 -> true             SURVIVED
L6  branch              b == 0 -> false            KILLED
L6  conditional         == -> !=                   KILLED
L6  conditional         == -> <                    KILLED
L6  conditional         == -> <=                   SURVIVED
L6  conditional         == -> >                    KILLED
L6  conditional         == -> >=                   SURVIVED
L6  boundary_value      b == 0 -> b == 1           KILLED
L7  boundary_value      return 0 -> return 1       SURVIVED
L7  string_literal      "b must not be zero" -> "" SURVIVED
L9  arithmetic          / -> +  / -> -  / -> *  / -> %   SURVIVED x4
L5, L6  empty_block     {...} -> {}                NOT_VIABLE x2

Killed 5   Survived 9   Not viable 2
all 5 kills: panic: integer divide by zero, not an assertion
coverage 100%        mutation score 5/14 = 35.7%
```

Specifying the behavior

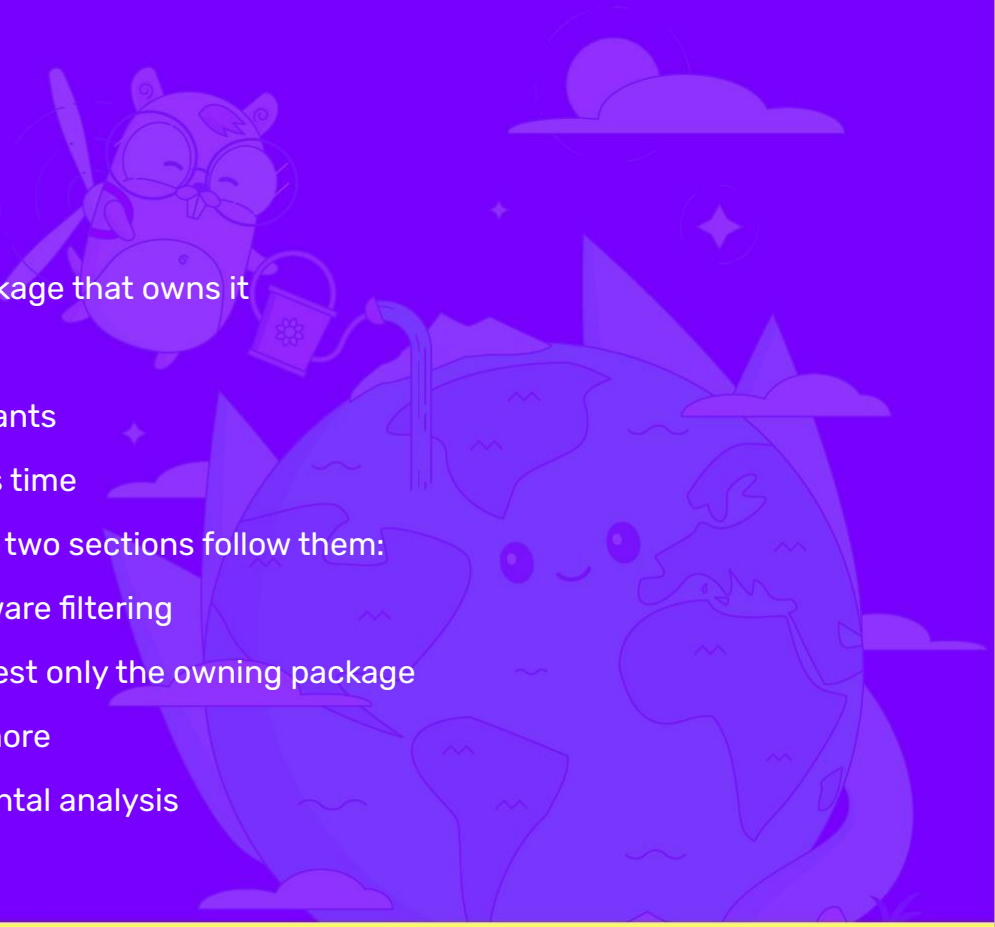
```go
func TestDiv(t *testing.T) {
    tests := []struct {
        a, b, want int
        wantErr    bool
    }{
        {a: 6, b: 3, want: 2},        // kills a*b a+b a-b a%b, if true, b >= 0
        {a: 6, b: -3, want: -2},      // kills b <= 0
        {a: 6, b: 0, wantErr: true}, // kills return 1
    }
    for _, tt := range tests {
        got, err := Div(tt.a, tt.b)
        if (err != nil) != tt.wantErr || got != tt.want {
            t.Errorf("Div(%d, %d) = %d, %v", tt.a, tt.b, got, err)
        }
    }
}
// gomu run .  ->  Killed 13  Survived 1  Not viable 2  ->  92.9%
// the one survivor: "b must not be zero" -> ""
```

Three things to know, and cost



- Not every survivor needs a test. Is the exact error message part of the API? ("b must not…" → "")
- Equivalent mutants: some changes are unobservable (a * 1 → a / 1). Accept them.
- Don't aim for 100% KILLED. Goodhart applies here too.
- **Cost: every mutant = a go build + a go test. The whole second half is about this.**

What a run costs



- Every mutant = go build + go test of the package that owns it
- Div, 5 lines → 16 mutants
- examples/calculator.go, 50 lines → 75 mutants
- Cost = number of mutants × your test suite's time
- Four ways gomu reduces this cost. The next two sections follow them:
- ① generate fewer useless mutants: type-aware filtering
- ② make each mutant cheap: -overlay, and test only the owning package
- ③ run them in parallel: goroutines + semaphore
- ④ don't re-run what didn't change: incremental analysis

04

INSIDE GOMU: MAKING MUTANTS

github.com/sivchari/gomu: mutation testing built for Go and for CI.

- The pipeline
- Mutators on go/ast
- Type-aware filtering

The gomu pipeline

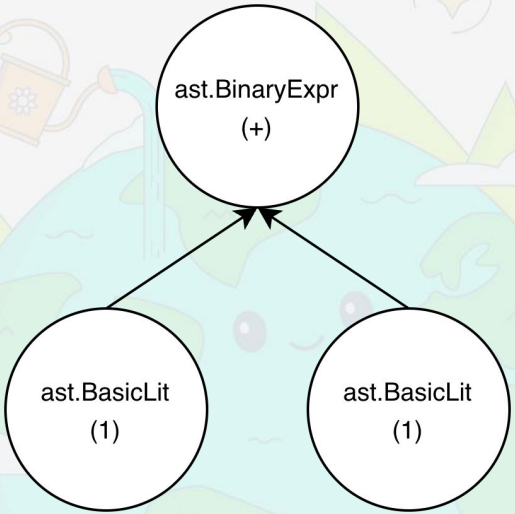


Only the Execute stage compiles or runs anything. Parse and Generate only work on the AST.

Start from the AST

- go/parser turns source into an AST; go/types adds type info
- A mutator is a function over ast.Node: no text patching, no regex

```
1 + 1  →  *ast.BinaryExpr{Op: token.ADD}
```



The Mutator interface

```
type Mutator interface {
    Name() string
    Description() string
    CanMutate(node ast.Node) bool
    Mutate(node ast.Node, fset *token.FileSet) []Mutant
    Apply(node ast.Node, mutant Mutant) bool
}

// Optional: mutations that replace the node itself (e.g. !ok -> ok)
type CursorApplier interface {
    ApplyWithCursor(node ast.Node, replace func(ast.Node), m Mutant) bool
}
```

Generating mutants

```
ast.Inspect(file, func(node ast.Node) bool {
    if _, ok := node.(*ast.ImportSpec); ok {
        return false // never mutate import paths
    }
    for _, m := range e.mutators {
        if !m.CanMutate(node) {
            continue
        }
        for _, mu := range m.Mutate(node, fset) {
            if tc.IsValidMutation(node, mu) { // go/types check
                mutants = append(mutants, mu)
            }
        }
    }
    return true
})
```

18 mutators (gomu run --list)

Operators

```
arithmetic
conditional
logical   (incl. !x → x)
bitwise
invert_negatives
remove_self_assignments
```

Values

```
boundary_value   (N → N±1)
string_literal
return
error_handling   (err → nil)
```

Control flow

```
branch   (cond → true/false)
loop_condition
break_continue
empty_block
switch
```

Removal

```
assignment_removal
expression_removal
statement_removal
```

Go-specific: errors

Go's error handling is explicit: the error is a value, and a value can be replaced with nil.

If return nil, err → return nil, nil survives, no test checks the error path.

error_handling targets any returned value whose type is error, not just the name err.

```
data, err := os.ReadFile(path)
if err != nil {
    return nil, err
}

// error_handling mutant
if err != nil {
    return nil, nil
}

// branch mutant
if false {
    return nil, err
}
```

Replacing nodes

You cannot remove ! by changing a field. You must replace the UnaryExpr with its operand.

CursorApplier gets a replace func that uses astutil.Cursor, which knows the parent node.

```
astutil.Apply(file, nil,
  func(c *astutil.Cursor) bool {
    pos := fset.Position(c.Node().Pos())
    if pos.Line == mu.Line && pos.Column == mu.Column {
        done = m.ApplyWithCursor(c.Node(), c.Replace, mu)
    }
    return !done
})

func (m *LogicalMutator) ApplyWithCursor(node ast.Node,
replaceFunc func(ast.Node), mutant Mutant) bool {
    if mutant.Type != logicalNotRemovalType { return false }
    expr, ok := node.(*ast.UnaryExpr)
    if !ok || expr.Op != token.NOT { return false }
    replaceFunc(expr.X)
    return true
}
```

Type-aware filtering

A mutant that doesn't compile costs a go build and says nothing about your tests.

gomu asks go/types first and drops invalid operators before anything is compiled. The compile step stays as a second check → NOT_VIABLE.

```
// string: only + is valid
s := first + last
//   + -> -     dropped

// error is an interface: only == and !=
if err != nil { ... }
//   != -> <    dropped

// int: every arithmetic op is valid
q := a / b
//   / -> *     kept
```

Adding a mutator: scaffold + go generate

```
$ go run ./cmd/scaffold -mutator=foo
  -> internal/mutation/foo.go
  -> internal/mutation/testdata/foo/input.go
  -> registry.go regenerated

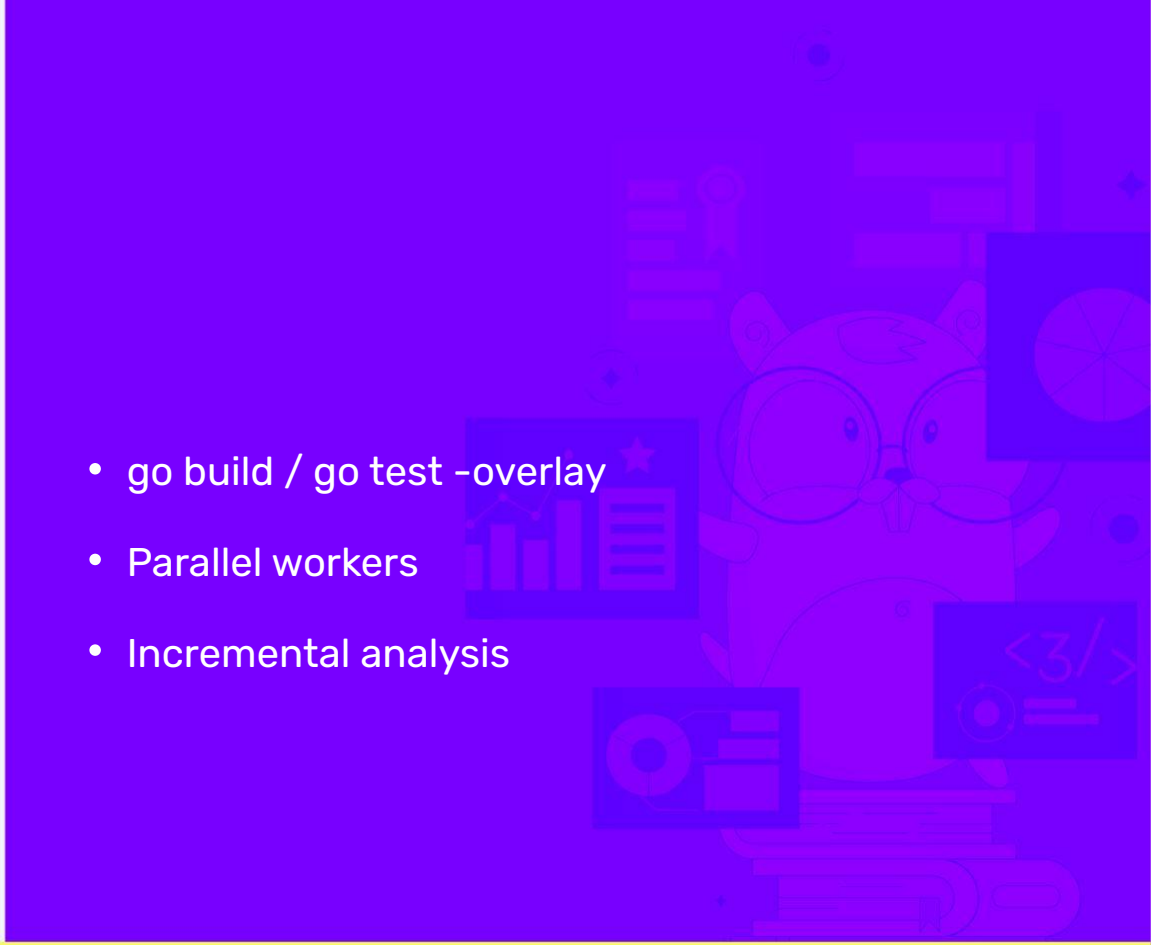

$ go test ./internal/mutation -run TestMutators/foo -update
  -> testdata/foo/*.golden
```

05

INSIDE GOMU: RUNNING MUTANTS

Thousands of mutants, zero edits to your working tree.

- go build / go test -overlay
- Parallel workers
- Incremental analysis



What is -overlay?

Added in Go 1.16. The go command reads file contents from a JSON replacement map instead of the real paths.

Designed for gopls: type-check the buffer you haven't saved yet. Exactly what a mutant that never exists in your repo needs.

```
// mutant_42/overlay.json
{
  "Replace": {
    "/repo/calc/div.go":
    "/tmp/gomu_overlay_x/mutant_42/div.go"
  }
}

$ go build -overlay=overlay.json .
$ go test  -overlay=overlay.json .
```

Why overlay?

- Your working tree is never modified, so a crash or Ctrl-C cannot leave a mutant in your repo
- One directory + overlay.json per mutant → no file locks between goroutines
- Only the mutated file is written, with no copy of the whole module
- While a mutant runs, its mutated file is on disk and you can read it

One mutant, end to end (simplified)

```go
func (e *Engine) runSingleMutation(m Mutant, timeout int) Result {
    ctx, _ := e.overlay.PrepareMutation(m) // mutated file + overlay.json
    defer e.overlay.CleanupMutation(ctx)

    // 1. does it compile?  go build -overlay=... .
    if err := e.checkCompilationWithOverlay(ctx); err != nil {
        return Result{Mutant: m, Status: StatusNotViable}
    }

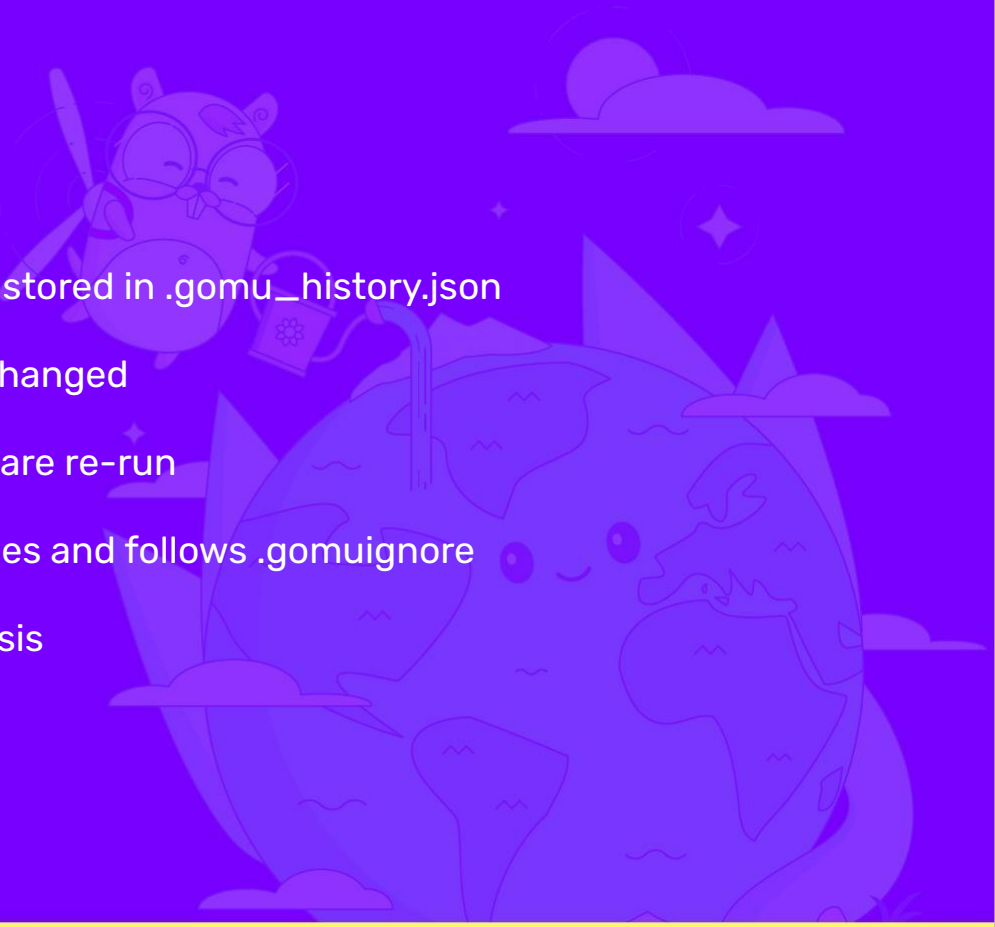
    // 2. go test -overlay=... . with a deadline
    //    exit != 0 -> KILLED, exit == 0 -> SURVIVED, deadline -> TIMED_OUT
    return e.runTestWithOverlay(ctx, m, timeout)
}
```

Parallelism: goroutines + a semaphore

```
results := make([]mutation.Result, len(mutants))
ch := make(chan indexedResult, len(mutants))
sem := make(chan struct{}, workers) // --workers

var wg sync.WaitGroup
for i, m := range mutants {
    wg.Go(func() {
        sem <- struct{}{}
        defer func() { <-sem }()
        ch <- indexedResult{i, e.runSingleMutation(m, timeout)}
    })
}
go func() { wg.Wait(); close(ch) }()
for r := range ch { results[r.index] = r.result }
```

Incremental analysis



- SHA-256 of each file and its _test.go, stored in .gomu_history.json
- Re-run a file only when it or its tests changed
- On PRs: only the files the PR changed are re-run
- Skips testdata/, vendor/, generated files and follows .gomuignore
- Inspired by PITest's incremental analysis

06

GOMU IN CI

Make test quality a number you can review and use as a gate on every PR.

- The CLI
- GitHub Actions & quality gate
- Practical tips

The CLI

```
$ go install github.com/sivchari/gomu/cmd/gomu@latest

$ gomu run --list                  # supported mutators
$ gomu run --dry-run ./calc        # mutations per file, not executed
$ gomu run ./calc                  # run it
$ gomu run --workers 8 --timeout 60 ./calc -- -short
$ gomu run --ci-mode --threshold 80 --output html
```

The quality gate

--ci-mode compares the score with --threshold. At or above: pass. Below: fail.
A failed gate exits 1, exactly like a failing test (--fail-on-gate, default true).
--fail-on-gate=false reports the number and never blocks. Start there.
Set the threshold from the score you have today, not the one you wish for.

```
$ gomu run --ci-mode --threshold 80 .
[1/1] calc.go (16 mutants) -> 5/16 killed

Killed:     5 (31.2%)
Survived:   9 (56.2%)
Not viable: 2 (12.5%)

Mutation Score: 35.7%
Quality Gate: FAILED (Score: 35.7%)
Reason: Mutation score 35.7% is below
        minimum threshold of 80.0%

$ echo $?
1
```

GitHub Actions

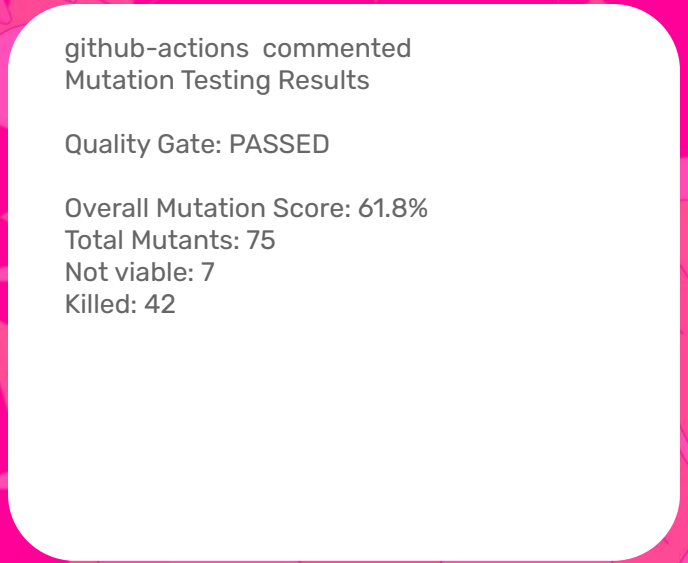
Incremental by default: only the files changed in the PR are mutated.

The action runs go test first, then gomu. It posts the score as a comment on the PR and fails the job below the threshold.

```yaml
on: pull_request
jobs:
  mutation:
    runs-on: ubuntu-latest
    permissions:
      contents: read
      pull-requests: write
    steps:
      - uses: actions/checkout@v5
      - uses: sivchari/gomu@v0.3.0
        with:
          go-version: '1.27'
          threshold: '80'
          comment-pr: 'true'
```

A quality gate on every PR

- The same gate, posted on the PR by the action
- The old gomu comment is replaced, not added below
- gomu's own CI runs gomu on itself, with the threshold set to its score on the day the gate was turned on. It only stops the score from going down.

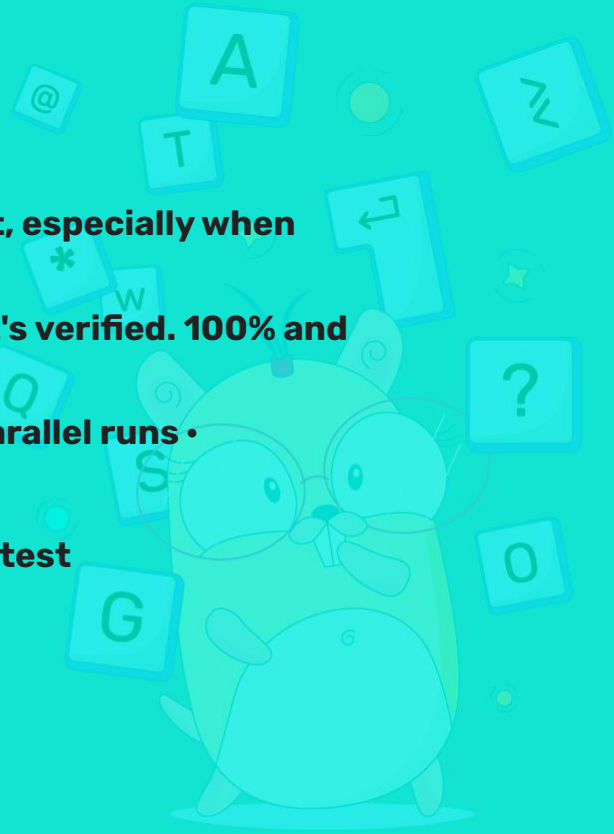


Practical tips

- Introduce it in three steps: local first, then PR comment only, then a gate
- Set the threshold at today's score, then raise it step by step
- Put the code you do not want tested in .gomuignore (generated files are skipped already)
- For each SURVIVED: add an assertion • accept an equivalent mutant • delete dead code
- **Give the SURVIVED list to your AI agent: “this line can change and no test notices” is a precise prompt**

SUMMARY

- **Green CI is only as trustworthy as the tests behind it, especially when agents write both**
- **Coverage says what ran. Mutation testing says what's verified. 100% and 35.7% were the same test.**
- **gomu: AST mutators • go/types filtering • -overlay parallel runs • incremental analysis • CI quality gate**
- **go install github.com/sivchari/gomu/cmd/gomu@latest**



Thank you!
Grazie!
ありがとう!



TAKUMA SHIBUYA

@sivchari
github.com/sivchari/gomu

