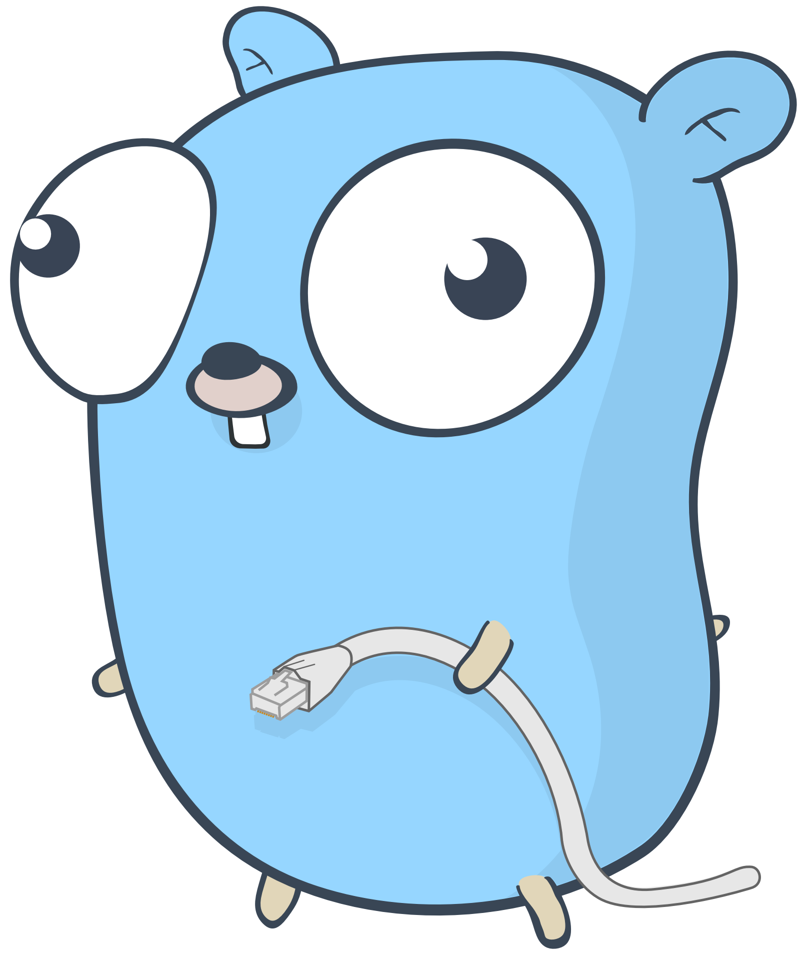
A Glimpse to **Profile-guided Optimization in Go**

Early practices of bringing PGO to production



Changkun Ou
changkun.de/s/gopgo

SIXT
Lisbon, Portugal
Nov. 28th, 2023

SIXT

About Me

Changkun Ou (@changkun)

- SIXT, Senior Engineer @ Pricing & Yield
- Engineering interests: Non-blocking optimizations / distributed consensus / graphics
- Active in Go Communities @golang, @fyne-io, @talkgo, @golang-design, …
- Email: hi@changkun.de

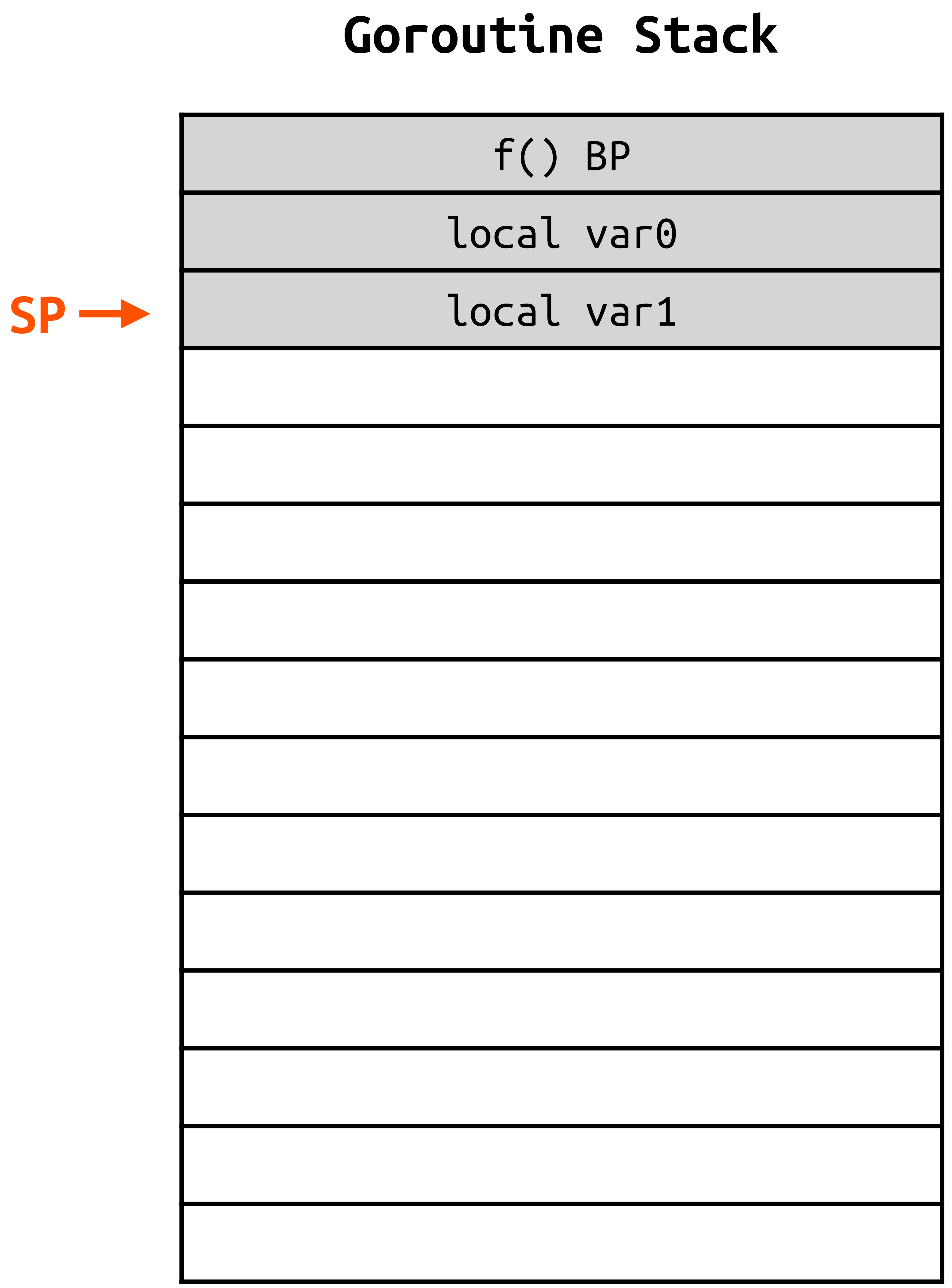
Agenda

- Invocation Overhead
- The Power of “Feedback Loop”
- Profile-guided Optimization (PGO) in Go
- Example and Applications
- Brining PGO to Production at Sixt
- Summary

Background

Invocation Overhead

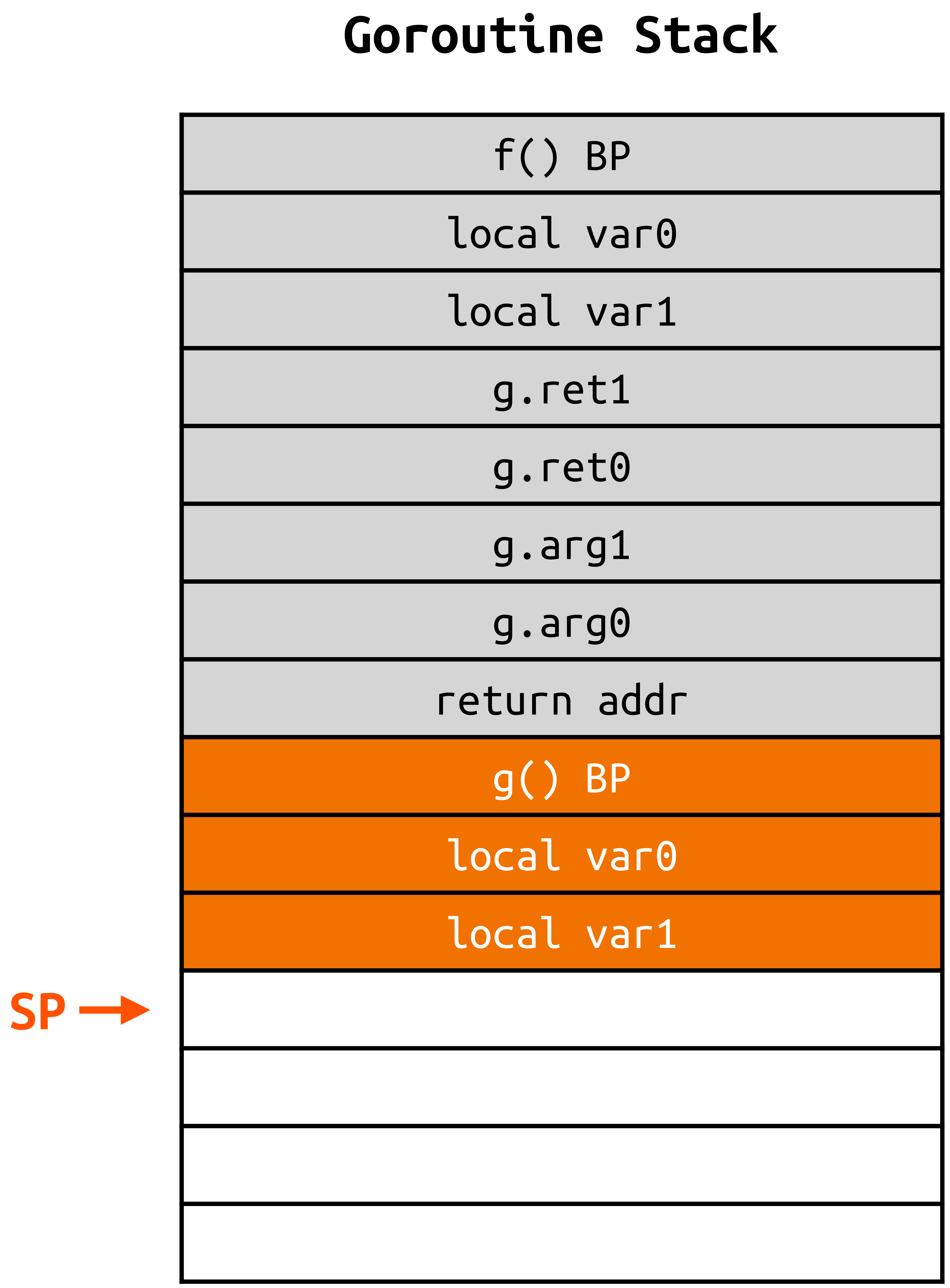
- Performing a function is not free but involve extra costs




```go
func f() {
    ...
    ret0, ret1 = g(arg0, arg1)
    ...
}
```

Invocation Overhead

- Performing a function is not free but involve extra costs
- When calling a function, arguments are copied on top of the stack

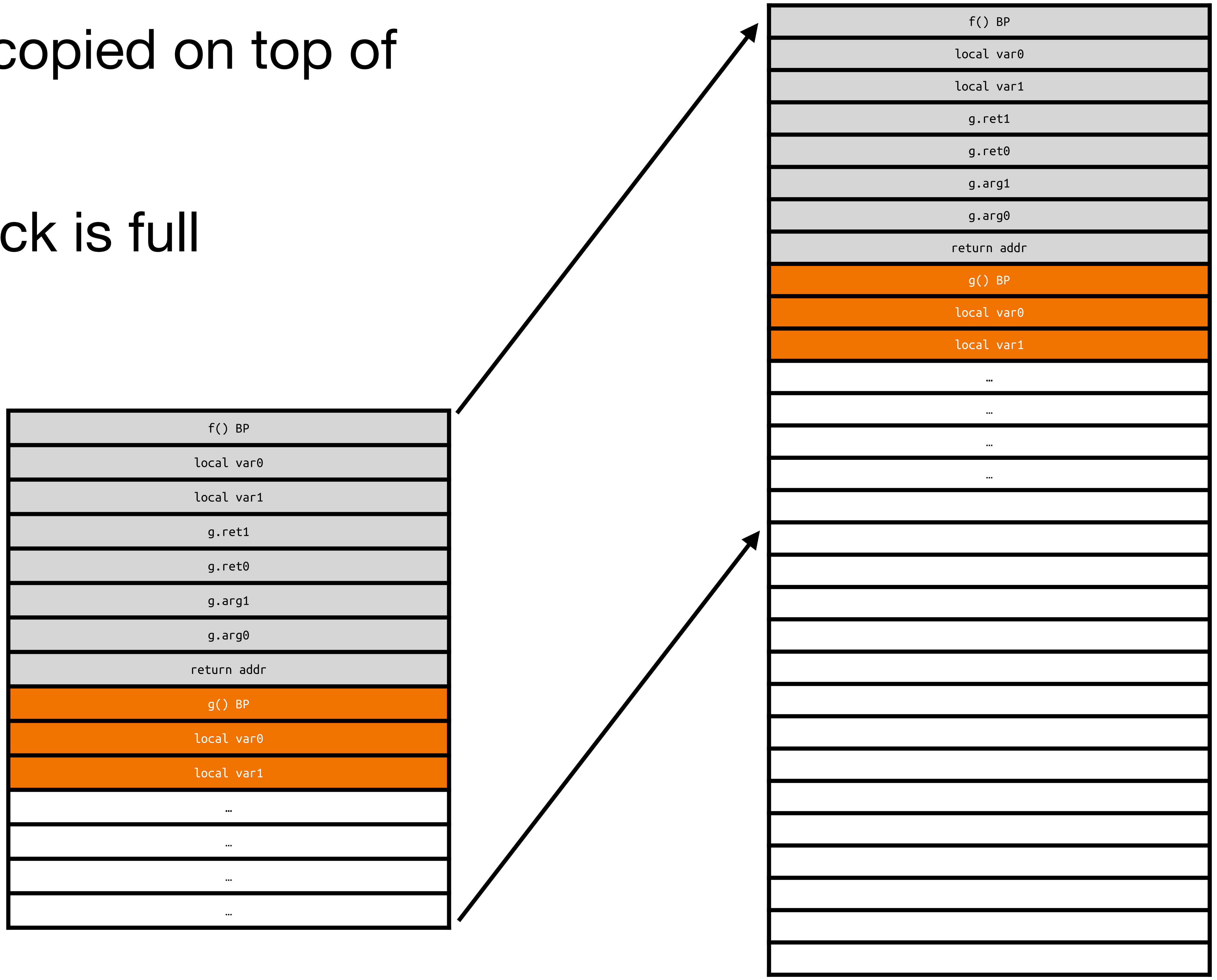



```go
func f() {
    ...
    ret0, ret1 = g(arg0, arg1)
    ...
}
```

Invocation Overhead

- Performing a function is not free but involve extra costs
- When calling a function, arguments are copied on top of the stack
- The entire stack maybe copied if the stack is full

Goroutine Stack




```go
func f() {
    ...
    ret0, ret1 = g(arg0, arg1)
    ...
}
```

Optimization: Inlining

- Inlining is a code transformation technique that replaces a function call (call site) with the body of the called function (callee)

```go
package main

func max(x, y int) int {
	if x > y {
		return x
	}
	return y
}

func main() {
	z := max(1, 2)
	println(z)
}
```

Optimization: Inlining

- Inlining is a code transformation technique that replaces a function call (call site) with the body of the called function (callee)

```go
package main

func max(x, y int) int {
    if x > y {
        return x
    }
    return y
}

func main() {
    z := max(1, 2)
    println(z)
}
```

→

```go
package main

func main() {
    var z int
    if 1 > 2 {
        z = 1
    } else {
        z = 2
    }
    println(z)
}
```

Optimization: Inlining

- Inlining is a code transformation technique that replaces a function call (call site) with the body of the called function (callee)

```go
package main

func max(x, y int) int {
    if x > y {
        return x
    }
    return y
}

func main() {
    z := max(1, 2)
    println(z)
}
```

→

```go
package main

func main() {
    var z int
    if 1 > 2 {
        z = 1
    } else {
        z = 2
    }
    println(z)
}
```

→

```go
package main

func main() {
    println(2)
}
```

`//go:noinline`, `-gcflags='-N -l'` can disable inlining

Optimization: Devirtualization

```go
func read(r io.Reader) []byte {
    buf := make([]byte, 1024)
    n, _ := r.Read(buf)
    return buf[:n]
}

func main() {
    f, err := os.Open("foo.txt")
    if err != nil { … }
    defer f.Close()

    fmt.Println(string(read(f)))
}
```

Optimization: Devirtualization

```go
func read(r io.Reader) []byte {
    buf := make([]byte, 1024)
    n, _ := r.Read(buf)
    return buf[:n]
}

func main() {
    f, err := os.Open("foo.txt")
    if err != nil { … }
    defer f.Close()

    fmt.Println(string(read(f)))
}
```

→

```go
func read(r io.Reader) (n []byte) {
    buf := make([]byte, 1024)
    if f, ok := r.(*os.File); ok {
        n, _ = f.Read(buf)
    } else {
        n, _ = f.Read(buf)
    }
    return buf[:n]
}

func main() {
    f, err := os.Open("foo.txt")
    if err != nil { … }
    defer f.Close()

    fmt.Println(string(read(f)))
}
```


SIXT

Optimization: Devirtualization

```go
func read(r io.Reader) []byte {
    buf := make([]byte, 1024)
    n, _ := r.Read(buf)
    return buf[:n]
}

func main() {
    f, err := os.Open("foo.txt")
    if err != nil { … }
    defer f.Close()

    fmt.Println(string(read(f)))
}
```

→

```go
func read(r io.Reader) (n []byte) {
    buf := make([]byte, 1024)
    if f, ok := r.(*os.File); ok {
        n, _ = f.Read(buf)
    } else {
        n, _ = f.Read(buf)
    }
    return buf[:n]
}

func main() {
    f, err := os.Open("foo.txt")
    if err != nil { … }
    defer f.Close()

    fmt.Println(string(read(f)))
}
```

→

```go
func main() {
    f, err := os.Open("foo.txt")
    if err != nil { … }
    defer f.Close()

    buf := make([]byte, 1024)
    n, _ := f.Read(buf)
    fmt.Println(string(buf[:n]))
}
```

Open Questions in Static Code Analysis

- Static code analysis cannot predict the runtime

```go
if (a < b) {
    foo()
    return
}

bar()
```

```go
switch n {
case 0:
    ...
case 1:
    ...
case ...
}
```

```go
for i := 0; i < n; i++ {
    ...
}
```

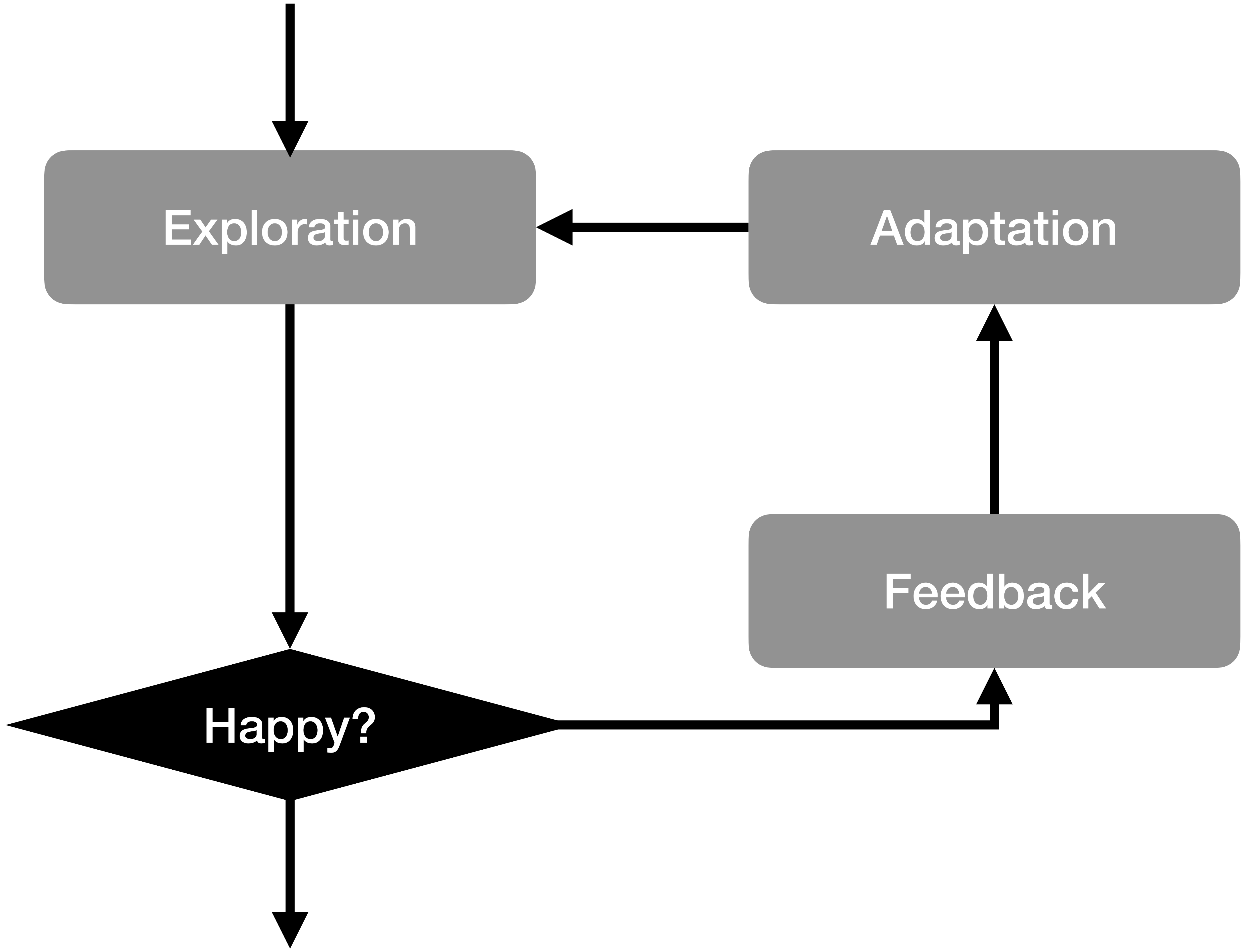
```go
type T struct {}
func (t *T) Bar() {}

type S struct {}
func (s S) Bar() {}

type Foo interface {
    Bar()
}

func foo(f Foo) {
    f.Bar()
}
```

How often is a < b?

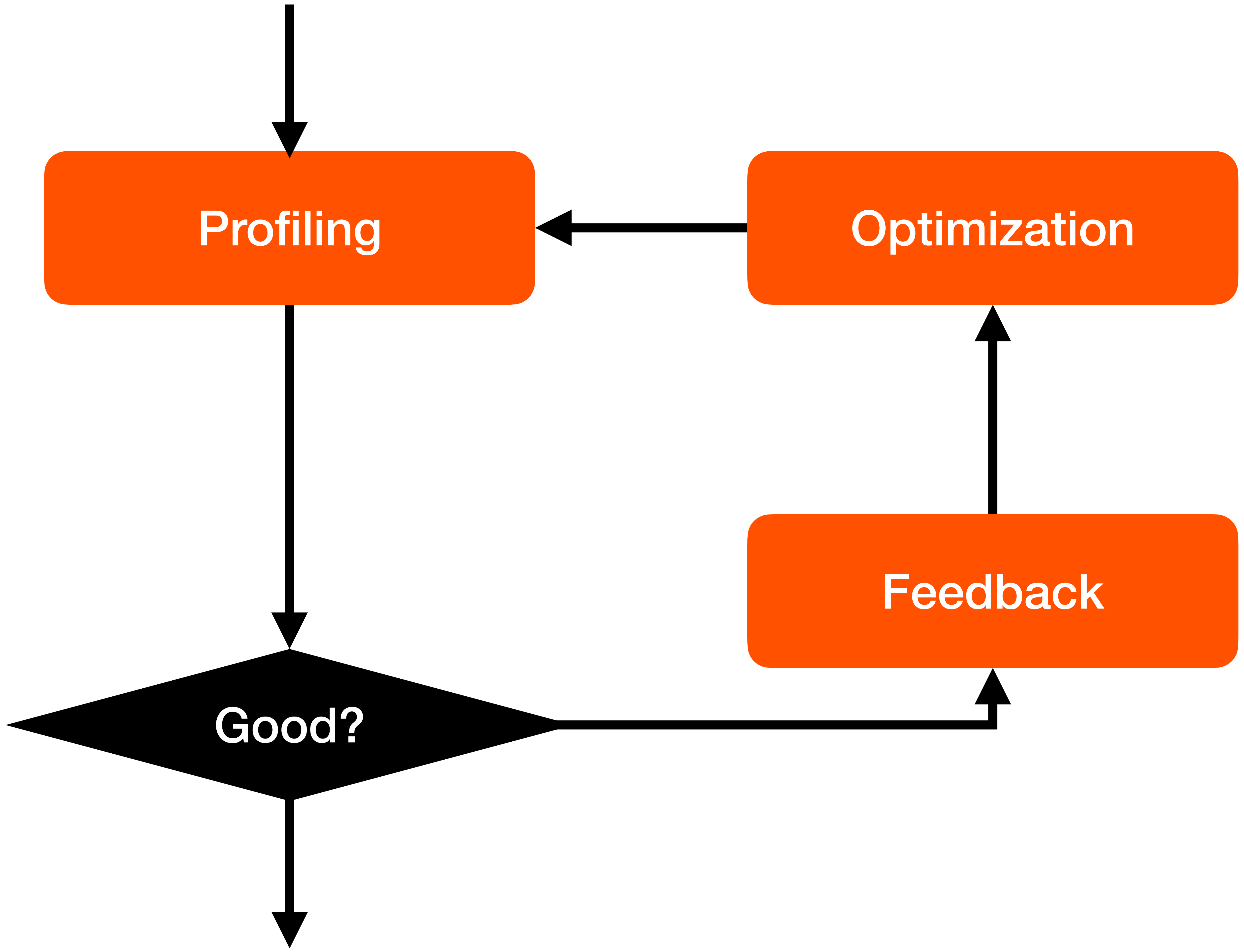
What's the typical value of n?

What's the typical type of parameter f?

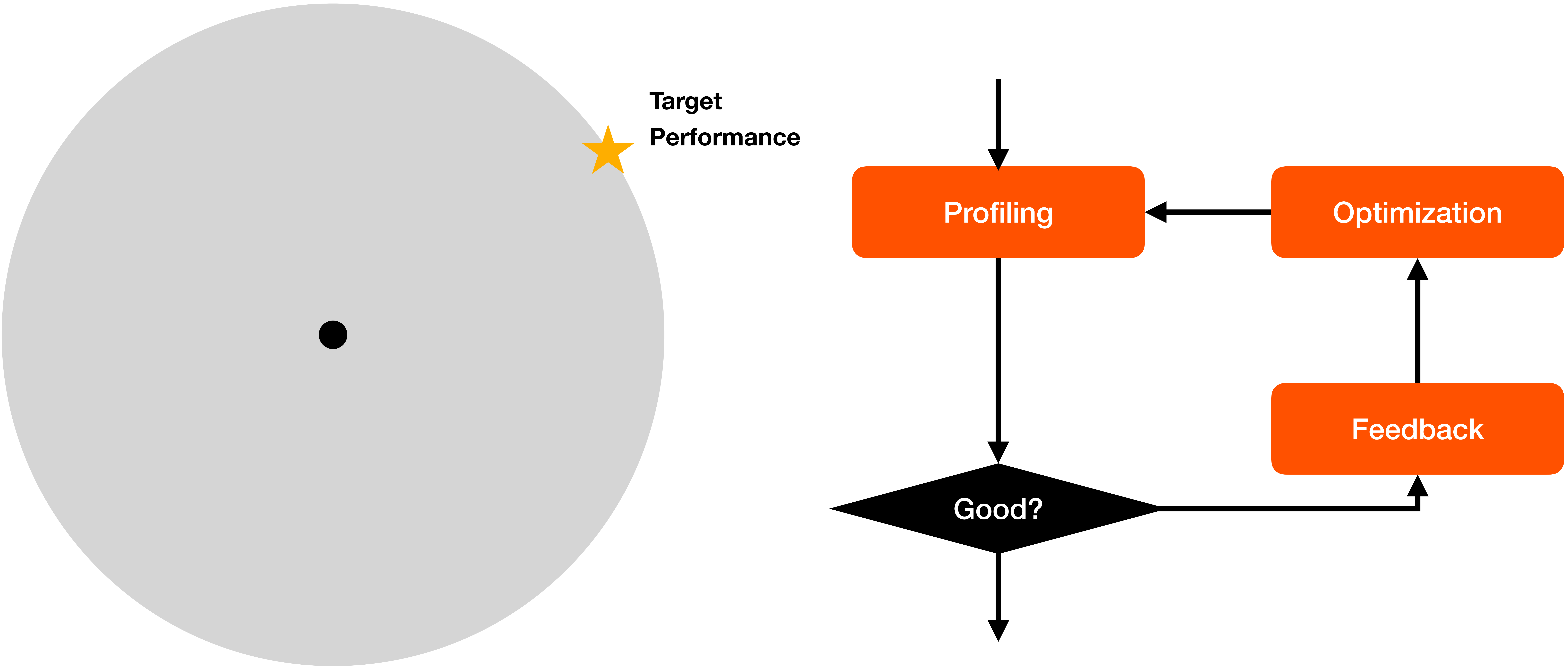
The Power of “Feedback Loop”



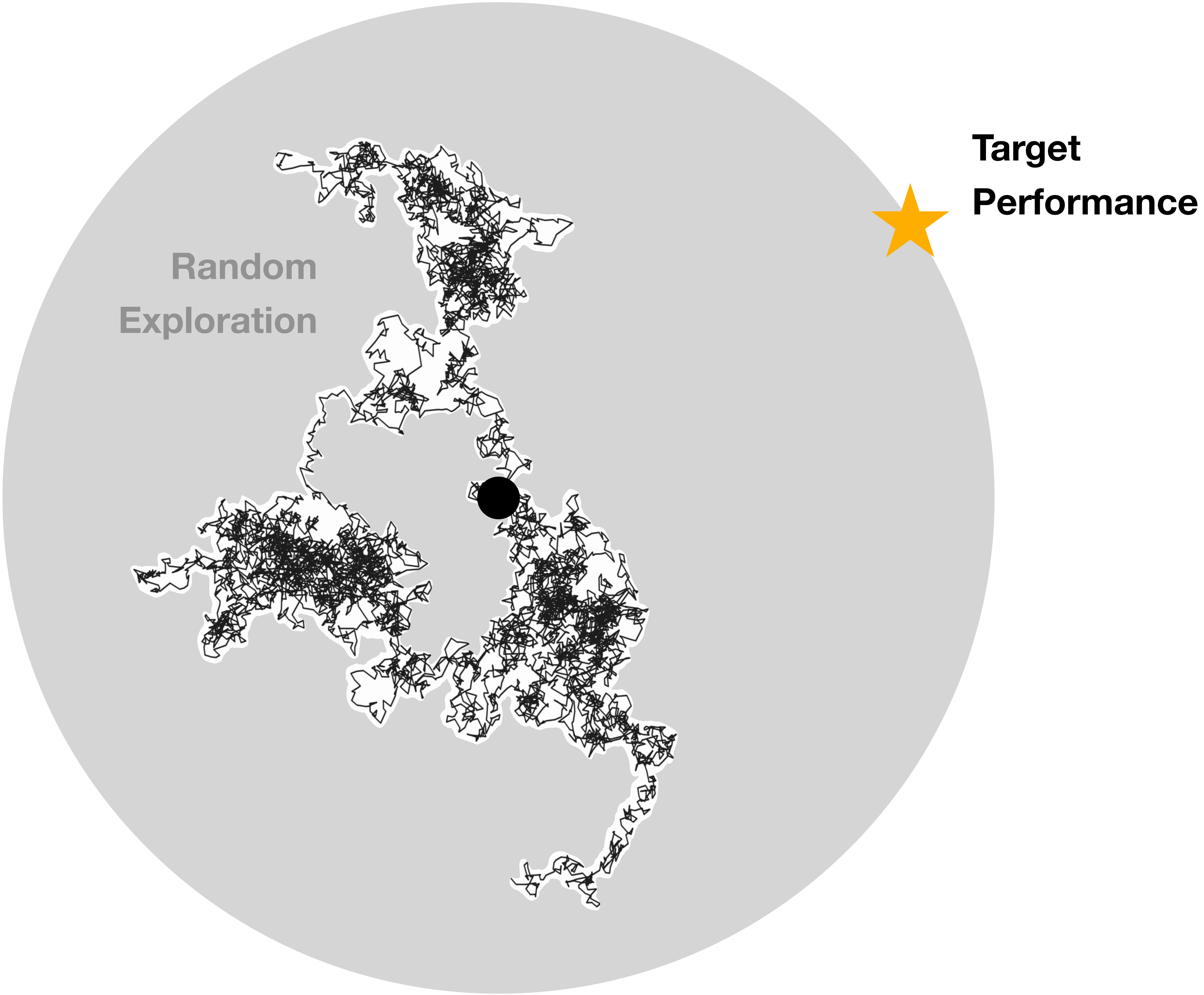
The Power of "Feedback Loop"

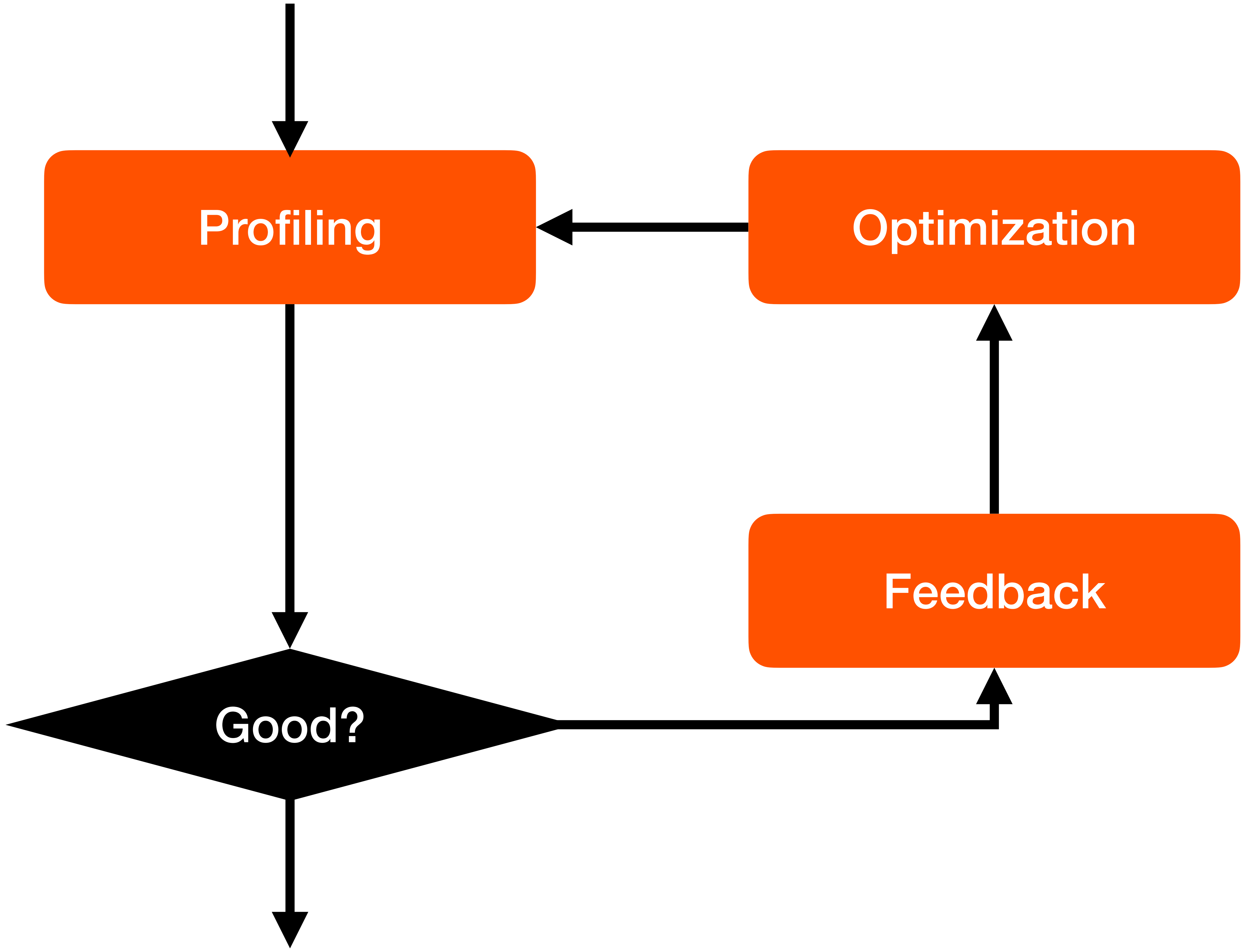


The Power of “Feedback Loop”

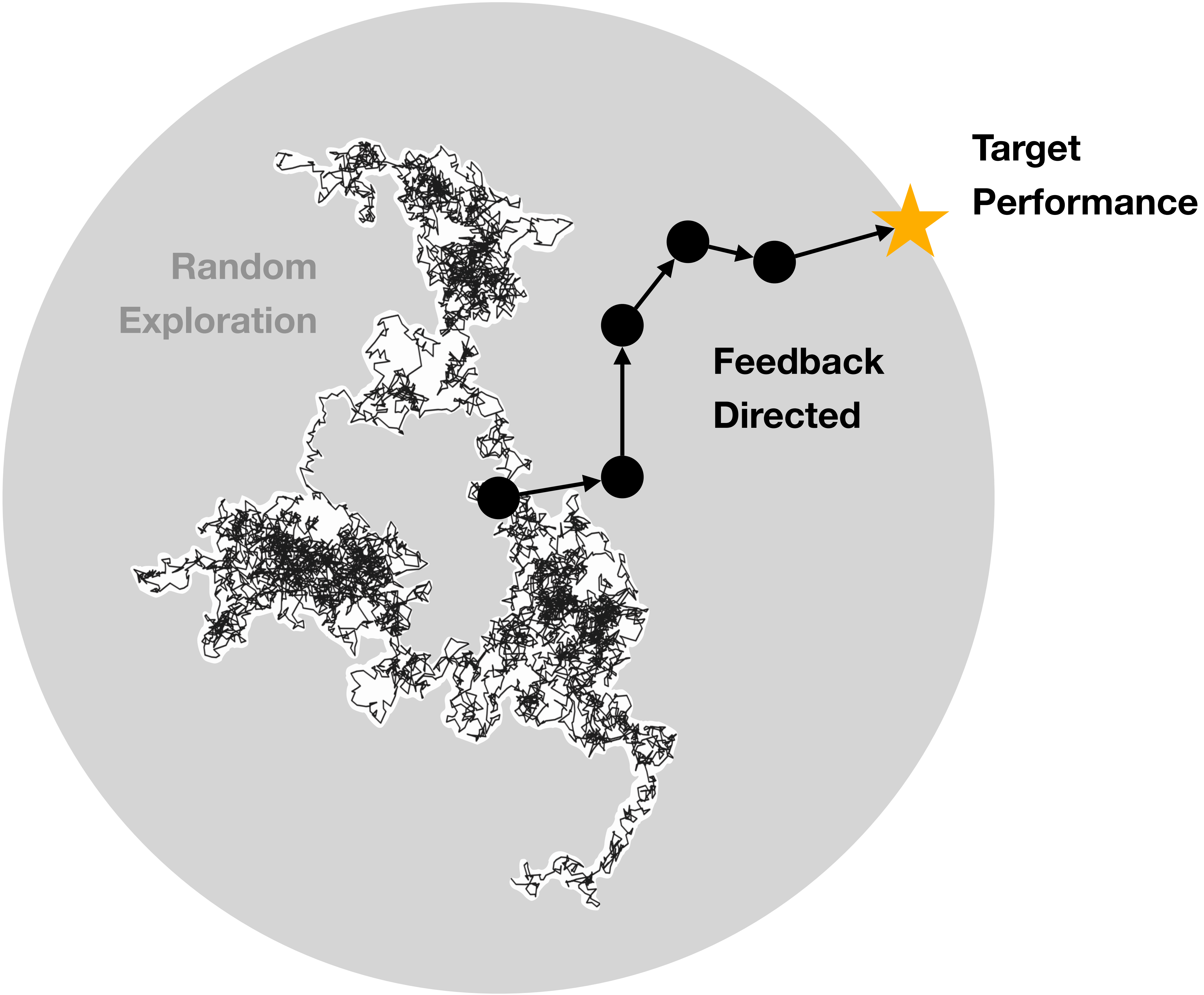


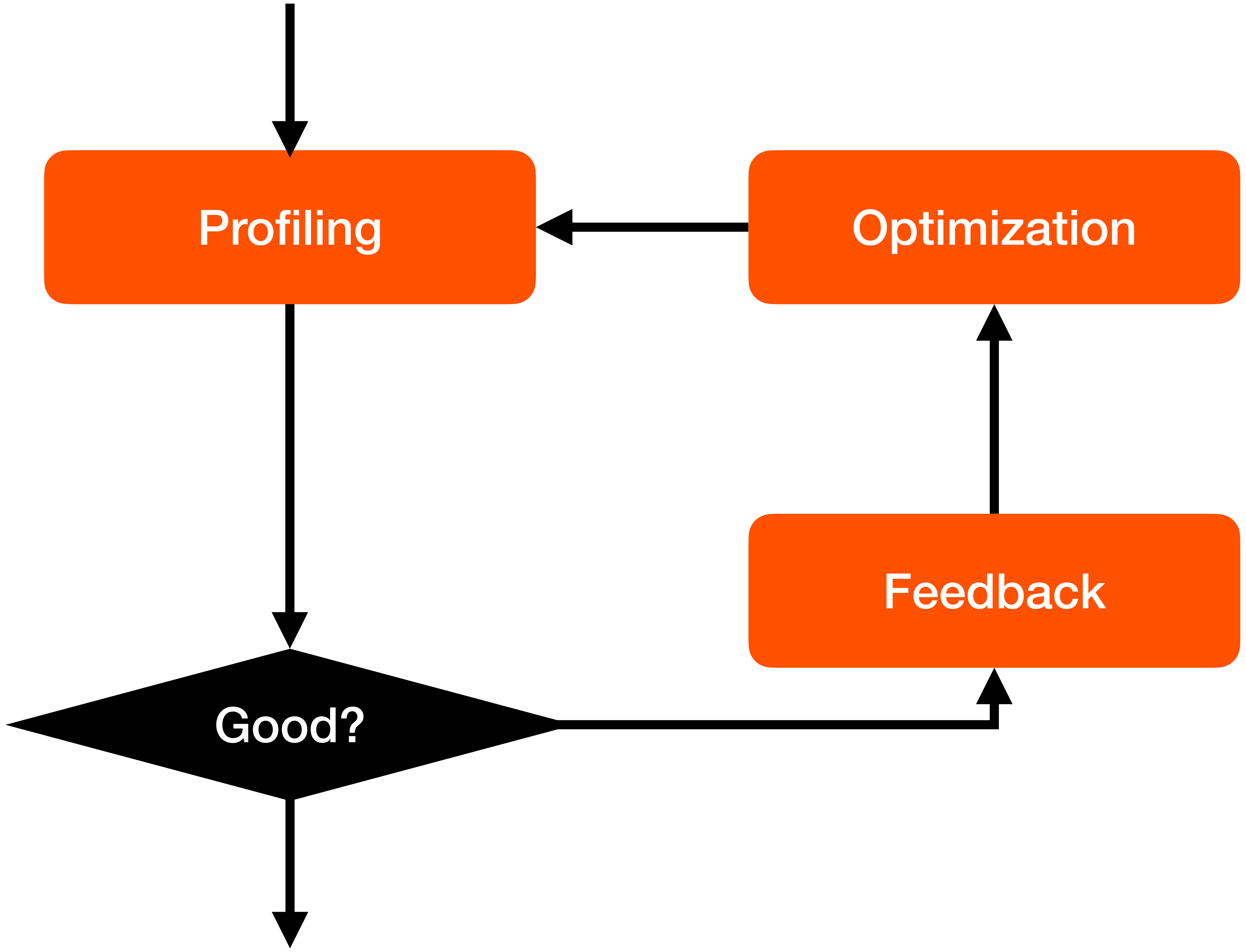
The Power of "Feedback Loop"





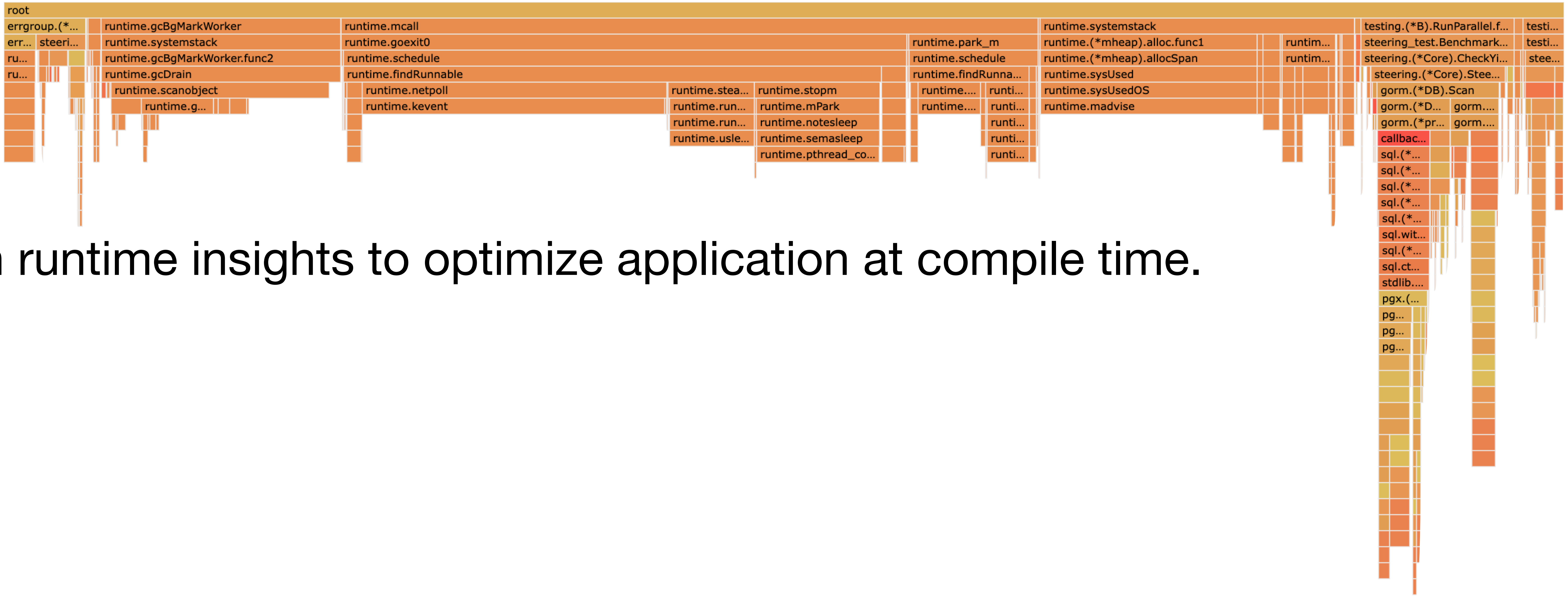
The Power of “Feedback Loop”





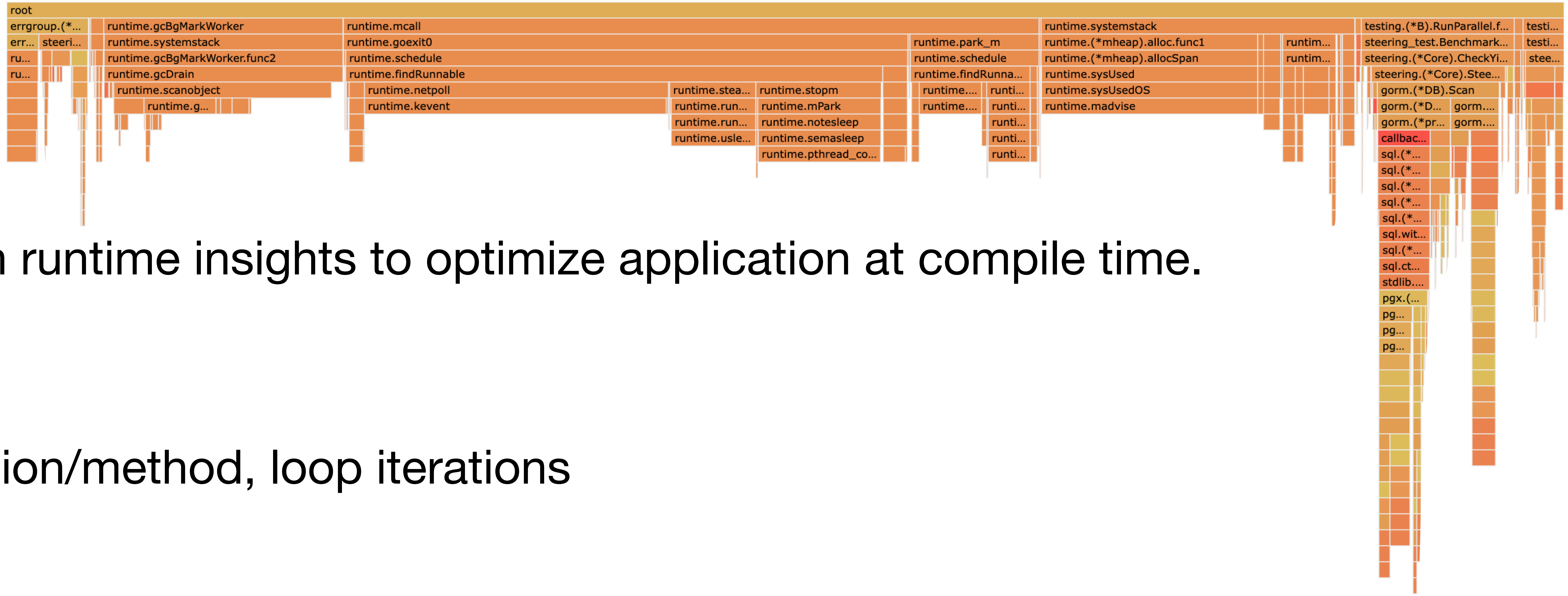
SIXT

Profiling



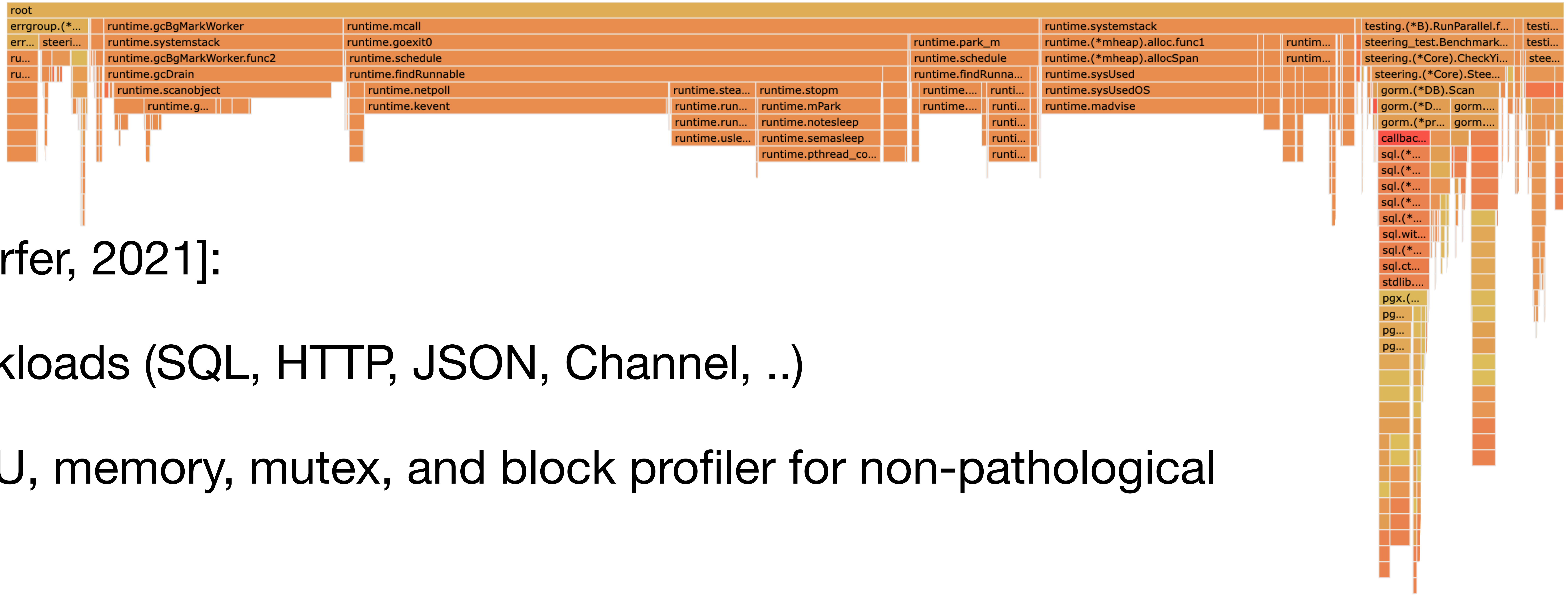
- Profiling brings data-driven runtime insights to optimize application at compile time.

Profiling



- Profiling brings data-driven runtime insights to optimize application at compile time.
- What’s profiled?
  - Number of calls to function/method, loop iterations
  - Branch probabilities
  - Memory consumption
  - Runtime activities
  - …

Profiling



Profiling overhead [Geisendörfer, 2021]:

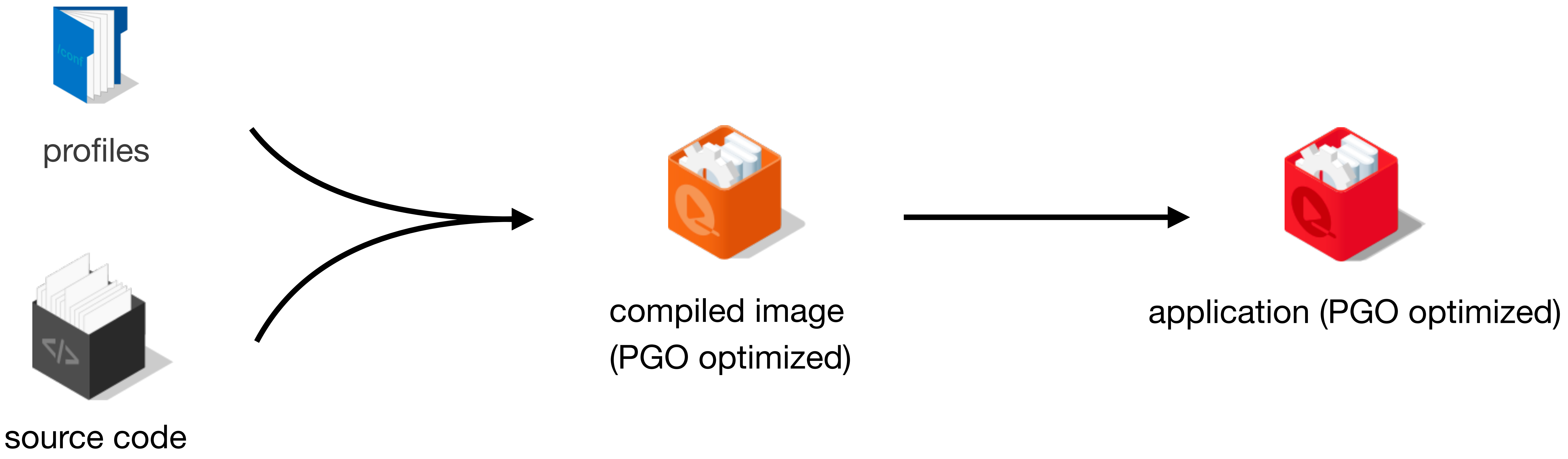
- Measured on different workloads (SQL, HTTP, JSON, Channel, ..)
- “Very low overhead for CPU, memory, mutex, and block profiler for non-pathological workloads”

Profile-guided Optimization (PGO)

- Unlike other static compiler optimization techniques, PGO requires user involvement to collect runtime profiles and feed them back into the build processes.

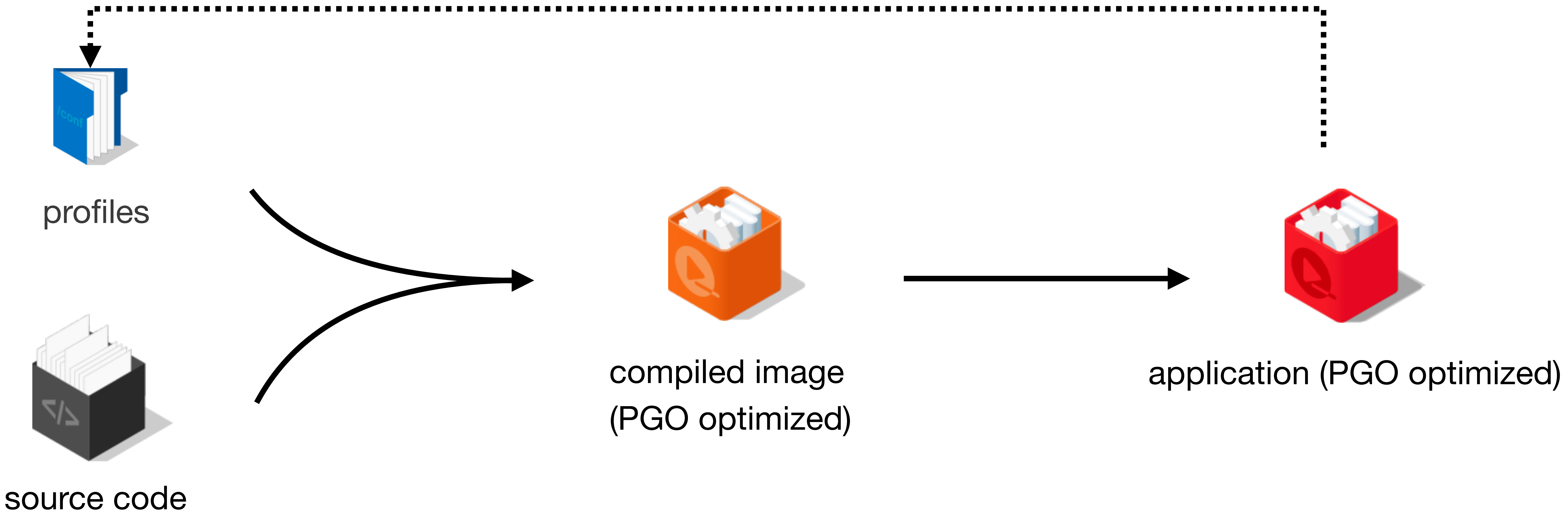
Profile-guided Optimization (PGO)

- Unlike other static compiler optimization techniques, PGO requires user involvement to collect runtime profiles and feed them back into the build processes.



Profile-guided Optimization (PGO)

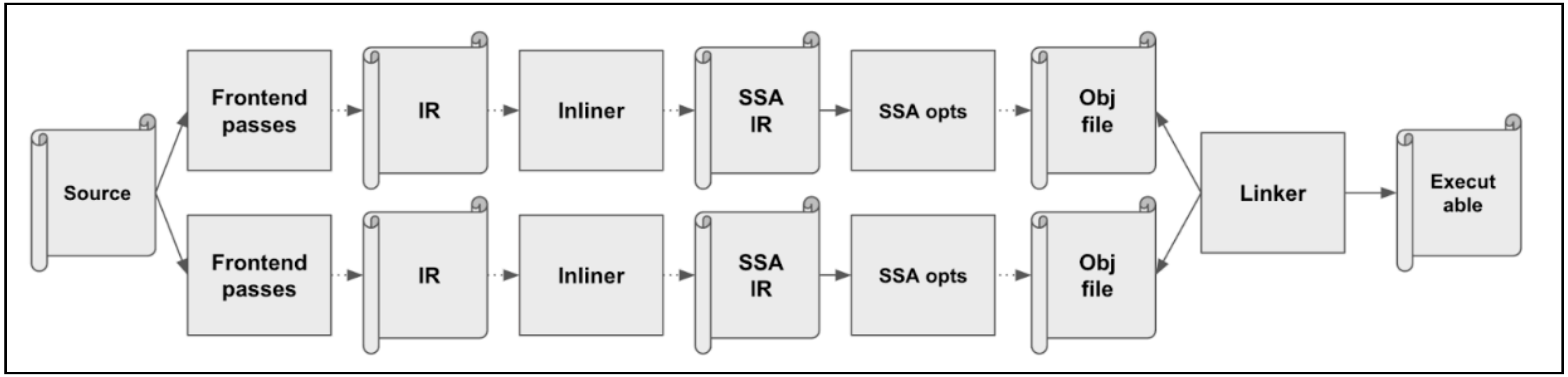
- Unlike other static compiler optimization techniques, PGO requires user involvement to collect runtime profiles and feed them back into the build processes.



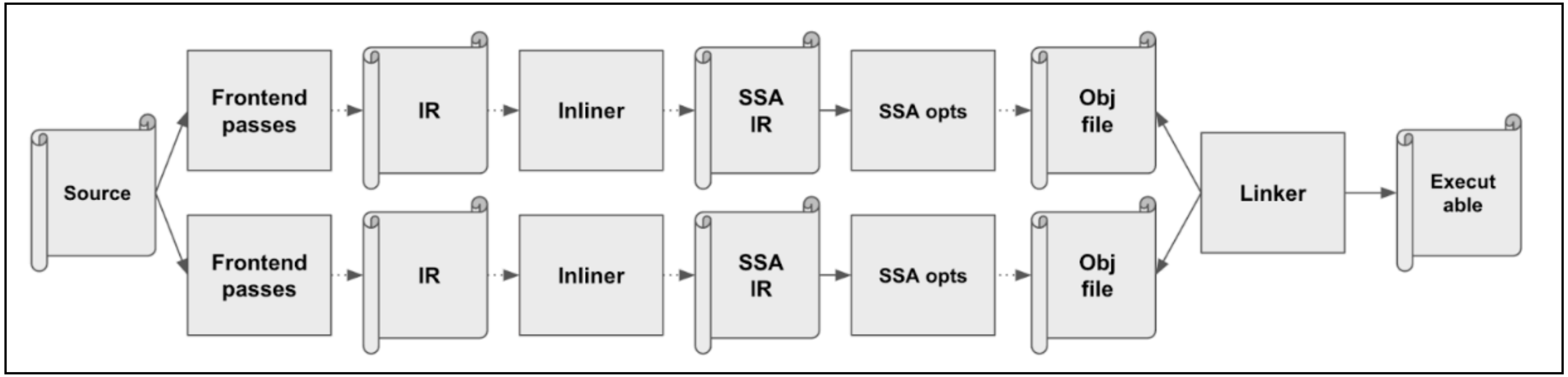
- PGO leverages runtime statistics regarding hot code paths and re-generate the same source code to execute favor and faster in those hot paths.

PGO in Go

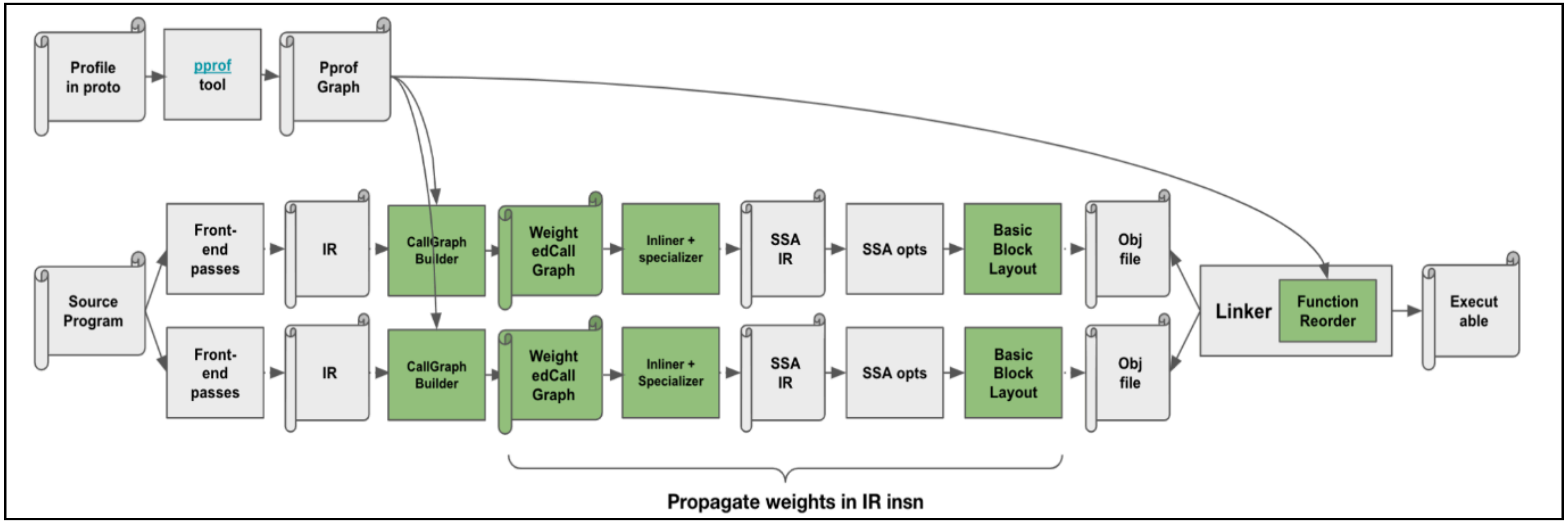
Go Code Compilation Process



Go Code Compilation Process



https://go.dev/design/55022-pgo



Enable Profiling in Go

To enable profiling in Go:

1. Using runtime/pprof or link net/http/pprof package or

```
import _ "net/http/pprof"
```

2. Run a http server:

```
go func() { http.ListenAndServe("localhost:6060", nil) }()
```

PGO in Go

To use PGO, there are essentially 3 steps:

1. Collect profiles:

```
$ wget -O cpu1.pprof http://service:6060//debug/pprof/profile?seconds=30

$ wget -O cpu2.pprof http://service:6060//debug/pprof/profile?seconds=30
```

2. Merge all profiles:

```
$ go tool pprof -proto cpu1.pprof cpu2.pprof > default.pgo
```

3. Build the binary using collected profile:

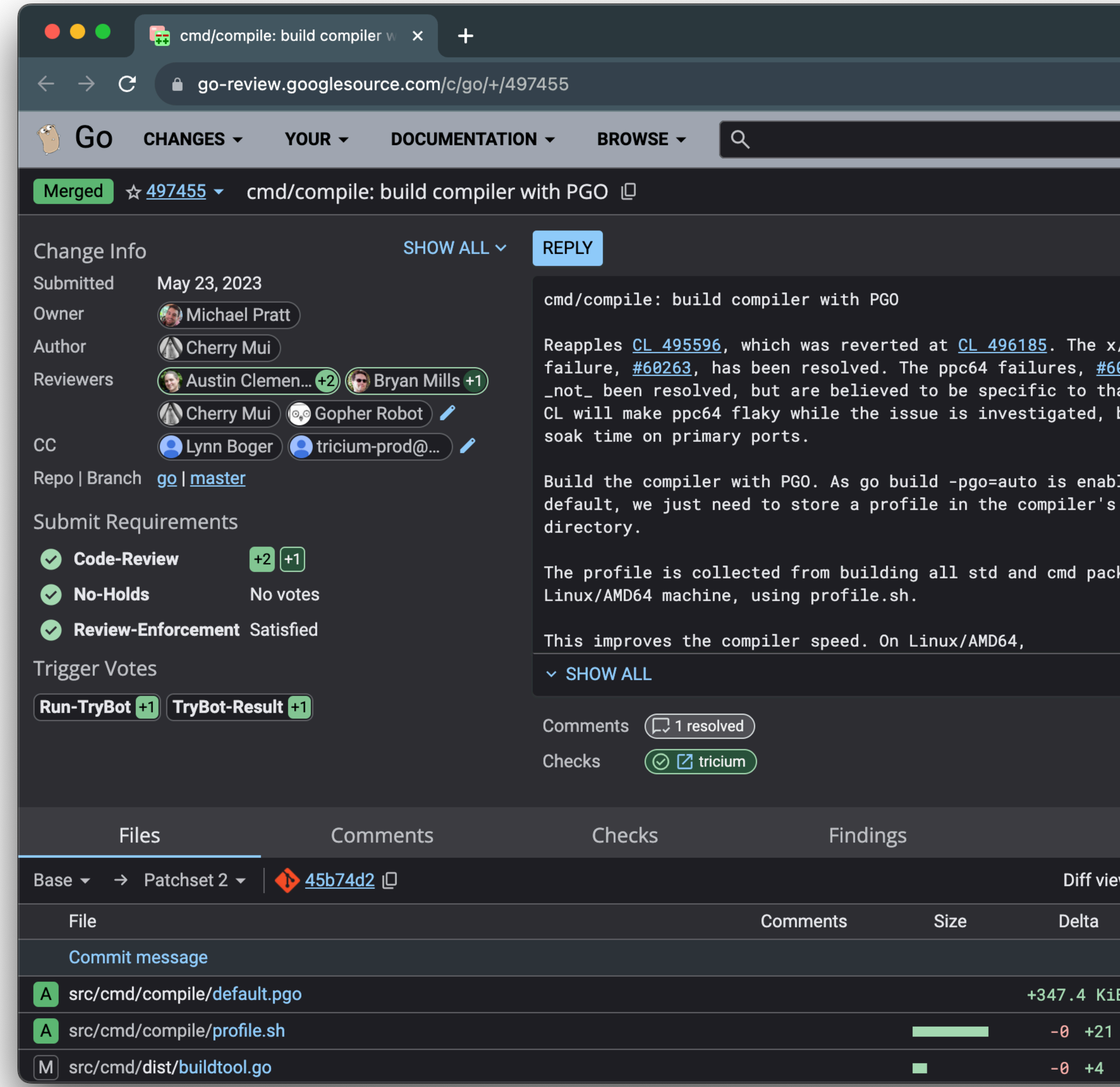
```
$ go build -pgo=default.pgo
```

As of Go 1.21, PGO in Go supports **inlining** and **devirtualization**.

Example: Optimize Go Compiler using PGO

As of Go 1.21, benchmarks for a representative set of Go programs show that building with PGO improves performance by around 2-7%.

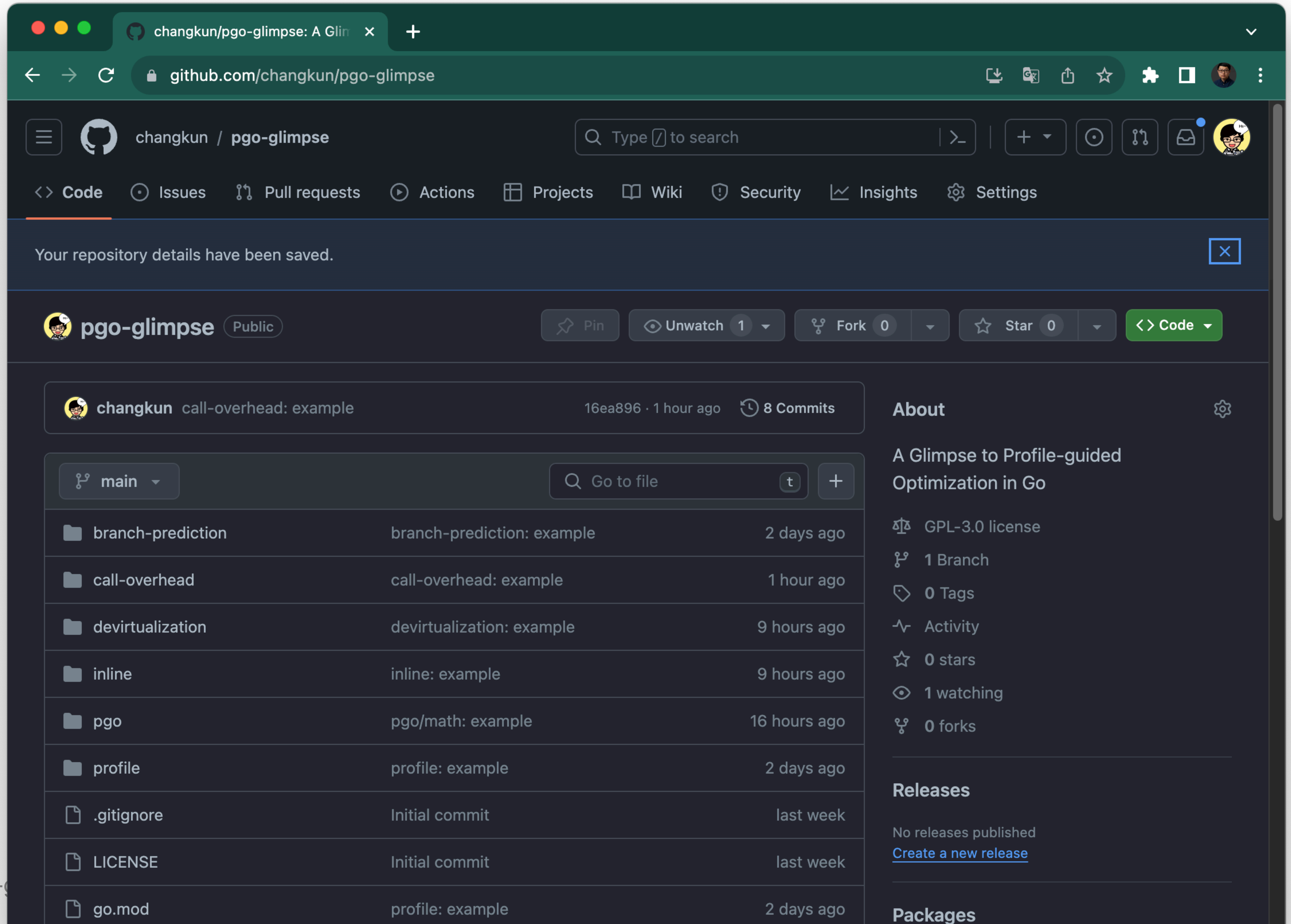
```
cmd/compile: build compiler with PGO

Build the compiler with PGO. As go build -pgo=auto is enabled by
default, we just need to store a profile in the compiler's
directory.

The profile is collected from building all std and cmd packages on
Linux/AMD64 machine, using profile.sh.

This improves the compiler speed. On Darwin/ARM64,
name          old time/op      new time/op      delta
Template        71.0ms ± 2%     68.3ms ± 2%   -3.90%  (p=0.000 n=20+20)
Unicode         71.8ms ± 2%     66.8ms ± 2%   -6.90%  (p=0.000 n=20+20)
GoTypes          444ms ± 1%      428ms ± 1%   -3.53%  (p=0.000 n=19+20)
Compiler        48.9ms ± 3%     45.6ms ± 3%   -6.81%  (p=0.000 n=20+20)
SSA              3.25s ± 2%      3.09s ± 1%   -5.03%  (p=0.000 n=19+20)
Flate           44.0ms ± 2%     42.3ms ± 2%   -3.72%  (p=0.000 n=19+20)
GoParser        76.7ms ± 1%     73.5ms ± 1%   -4.15%  (p=0.000 n=18+19)
Reflect          172ms ± 1%      165ms ± 1%   -4.13%  (p=0.000 n=20+19)
Tar             63.1ms ± 1%     60.4ms ± 2%   -4.24%  (p=0.000 n=19+20)
XML             83.2ms ± 2%     79.2ms ± 2%   -4.79%  (p=0.000 n=20+20)
[Geo mean]       127ms           121ms        -4.73%
```



https://go.dev/cl/497455

More Examples

- https://github.com/changkun/pgo-glimpse



Practices

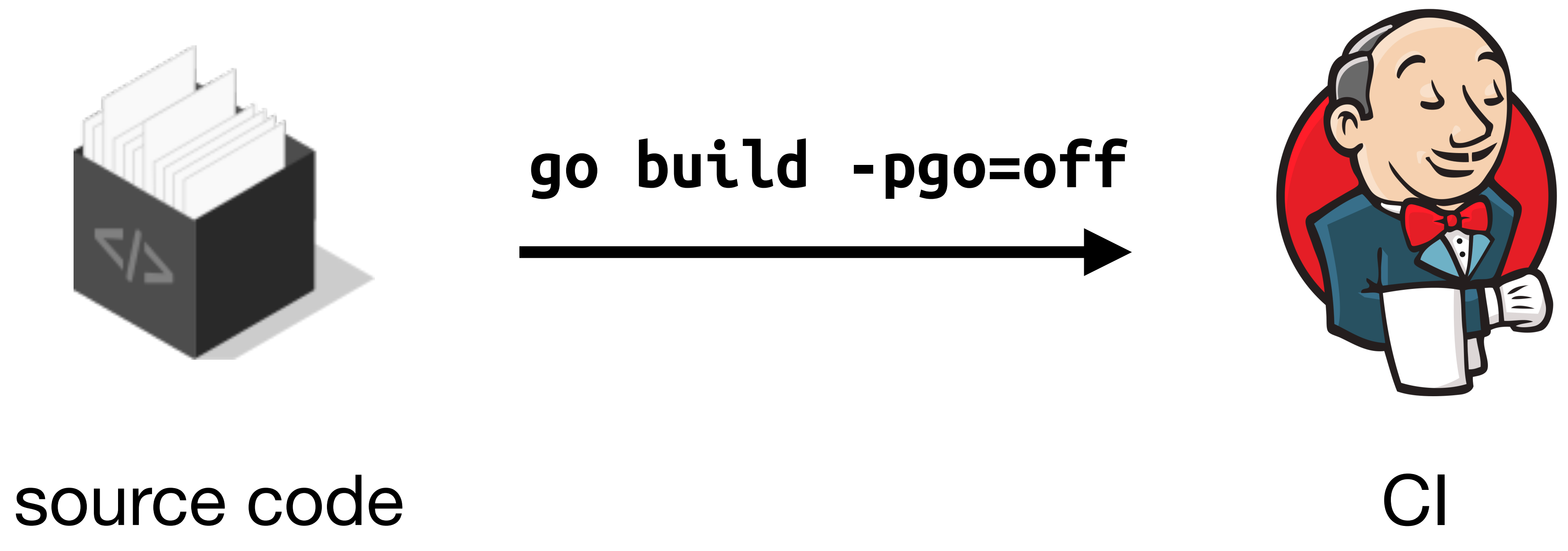
Challenges to Integrate PGO

- No platform infrastructure support :(
- No existing practices in the organization can bring strong arguments :(
- No existing practices can integrate PGO into CI/CD pipeline :(
- No baseline reference :(

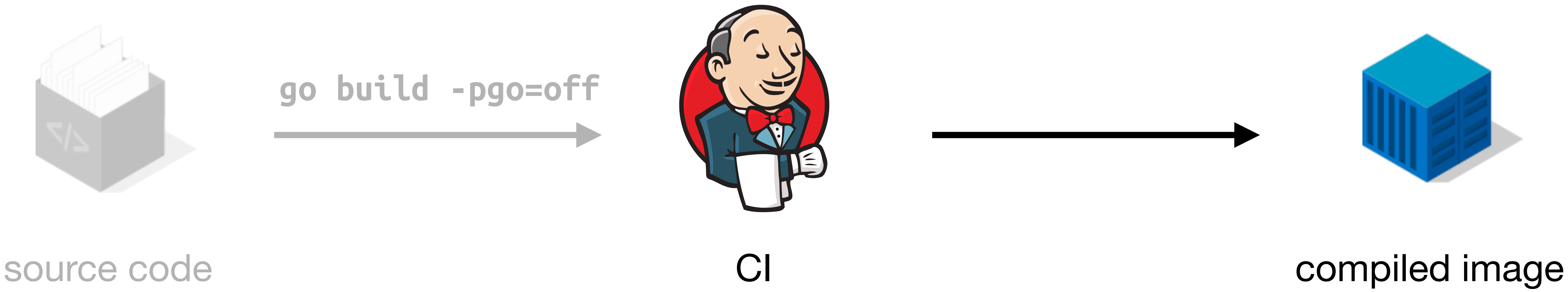
CI Release Workflow

source code

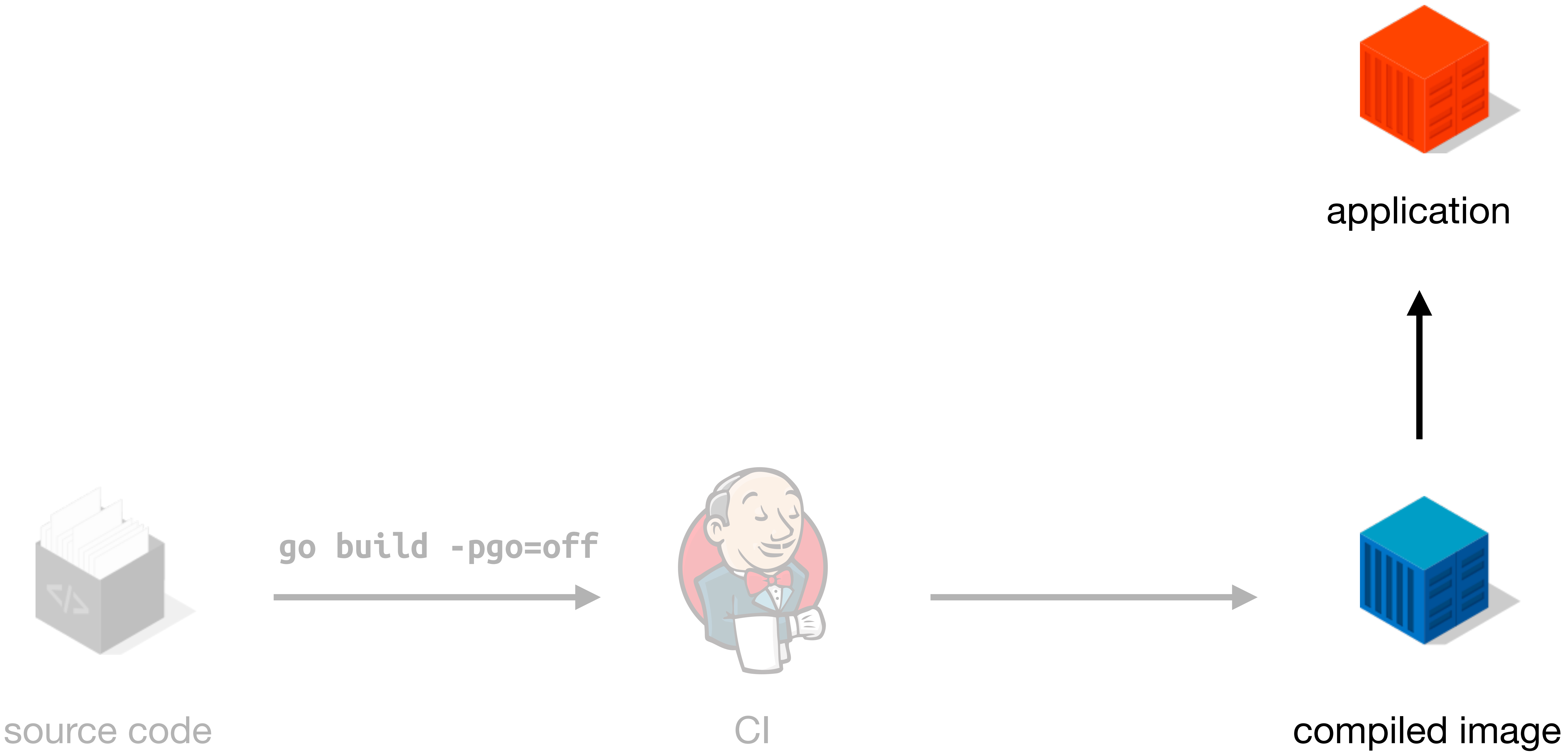
CI Release Workflow



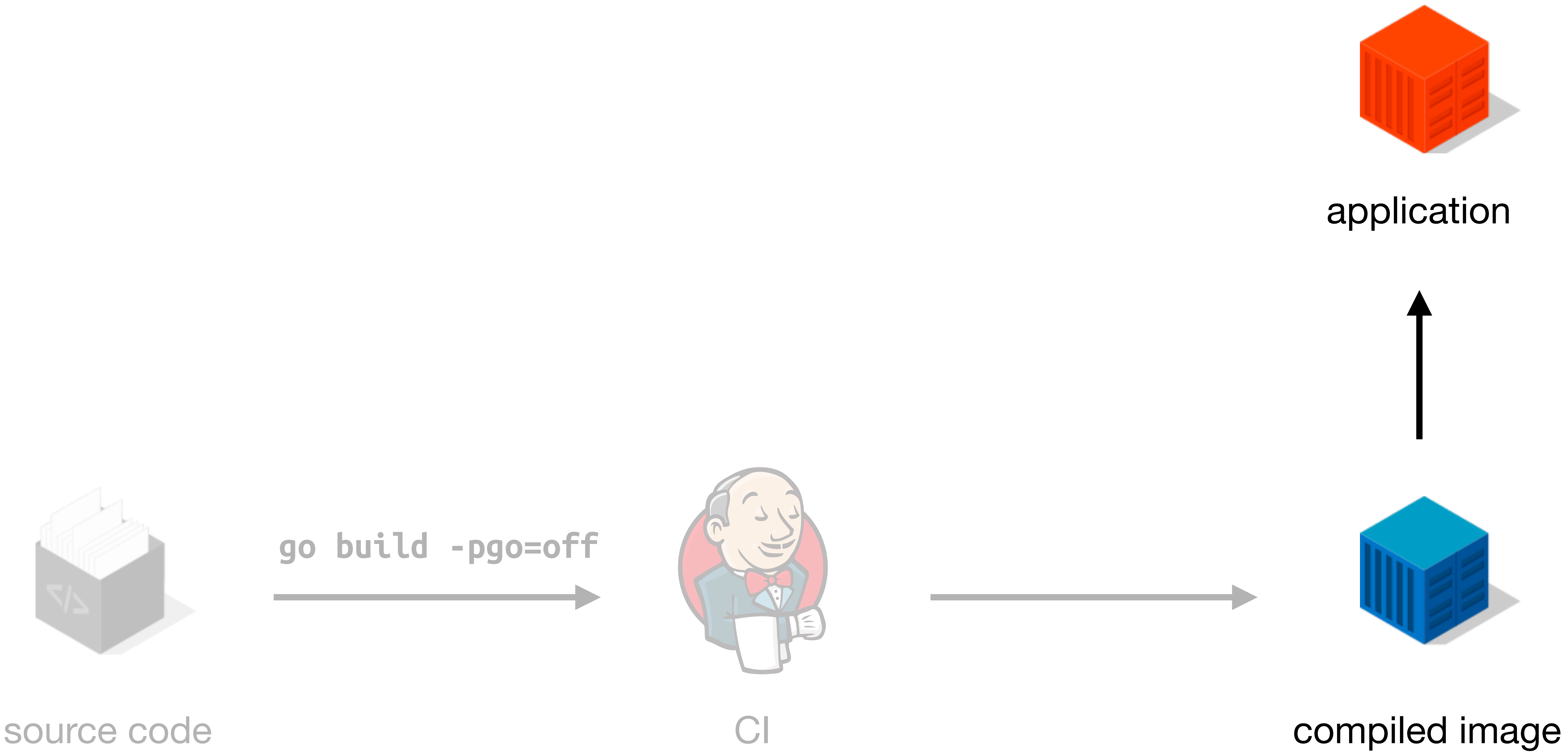
CI Release Workflow



CI Release Workflow

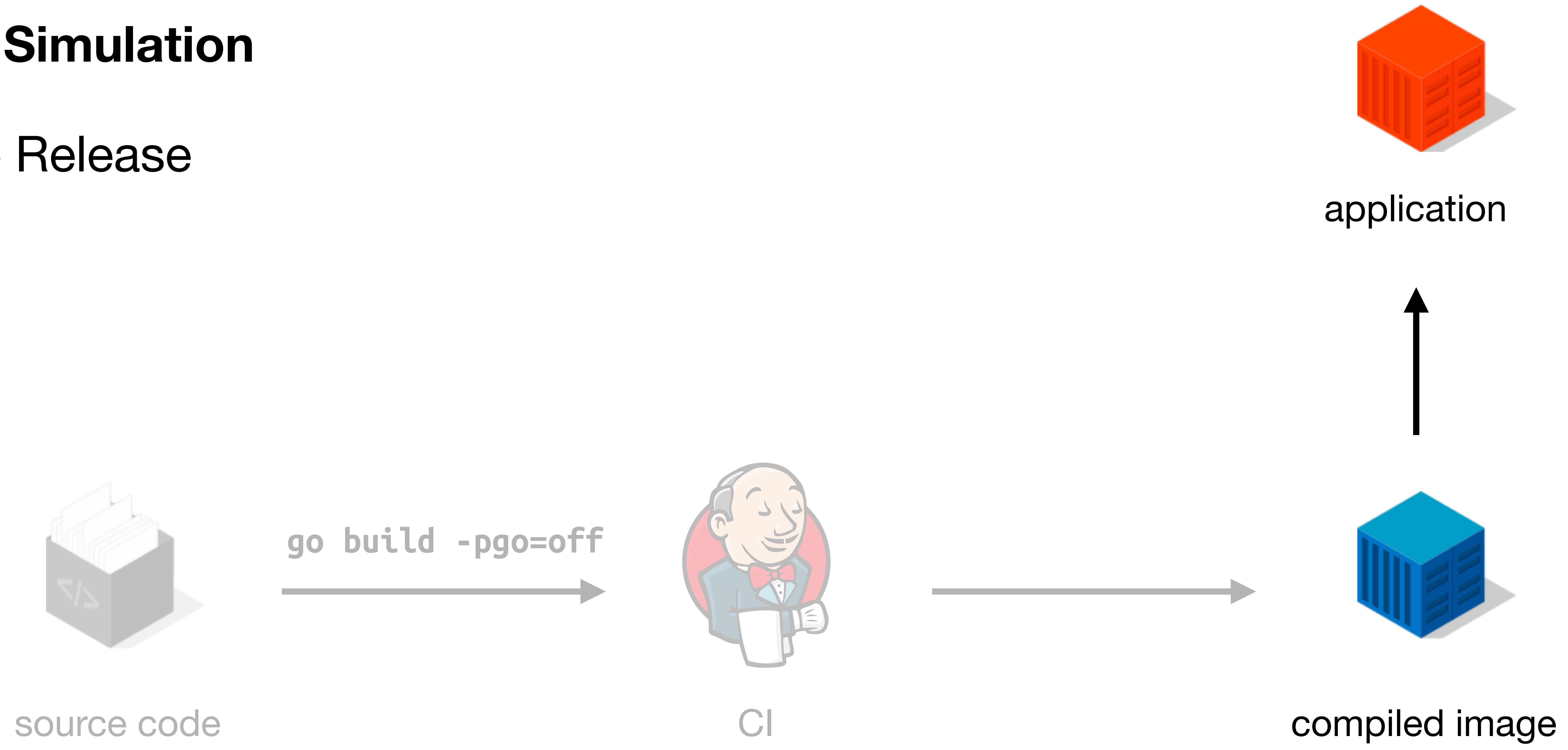


PGO Release Workflow

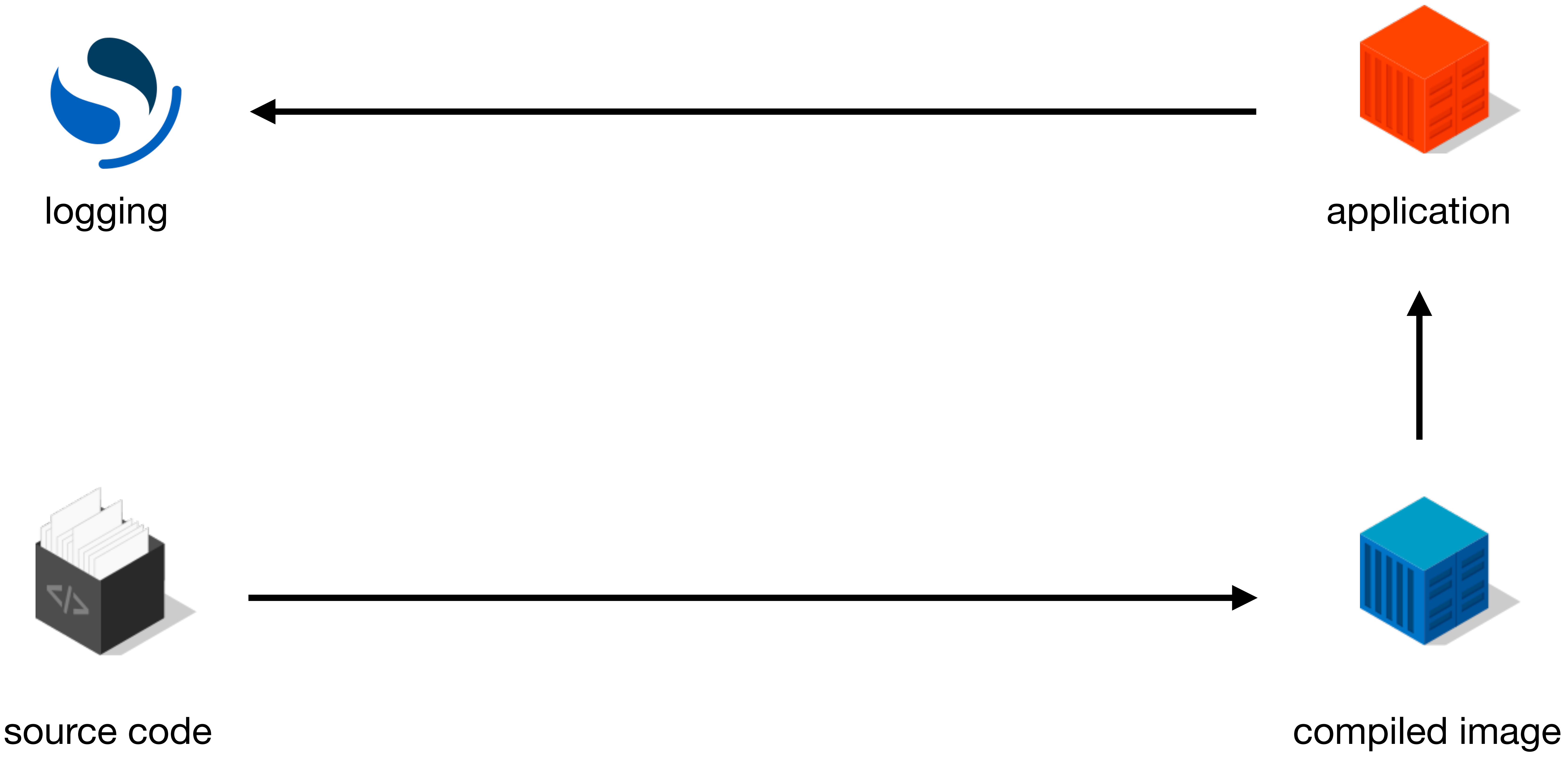


PGO Release Workflow

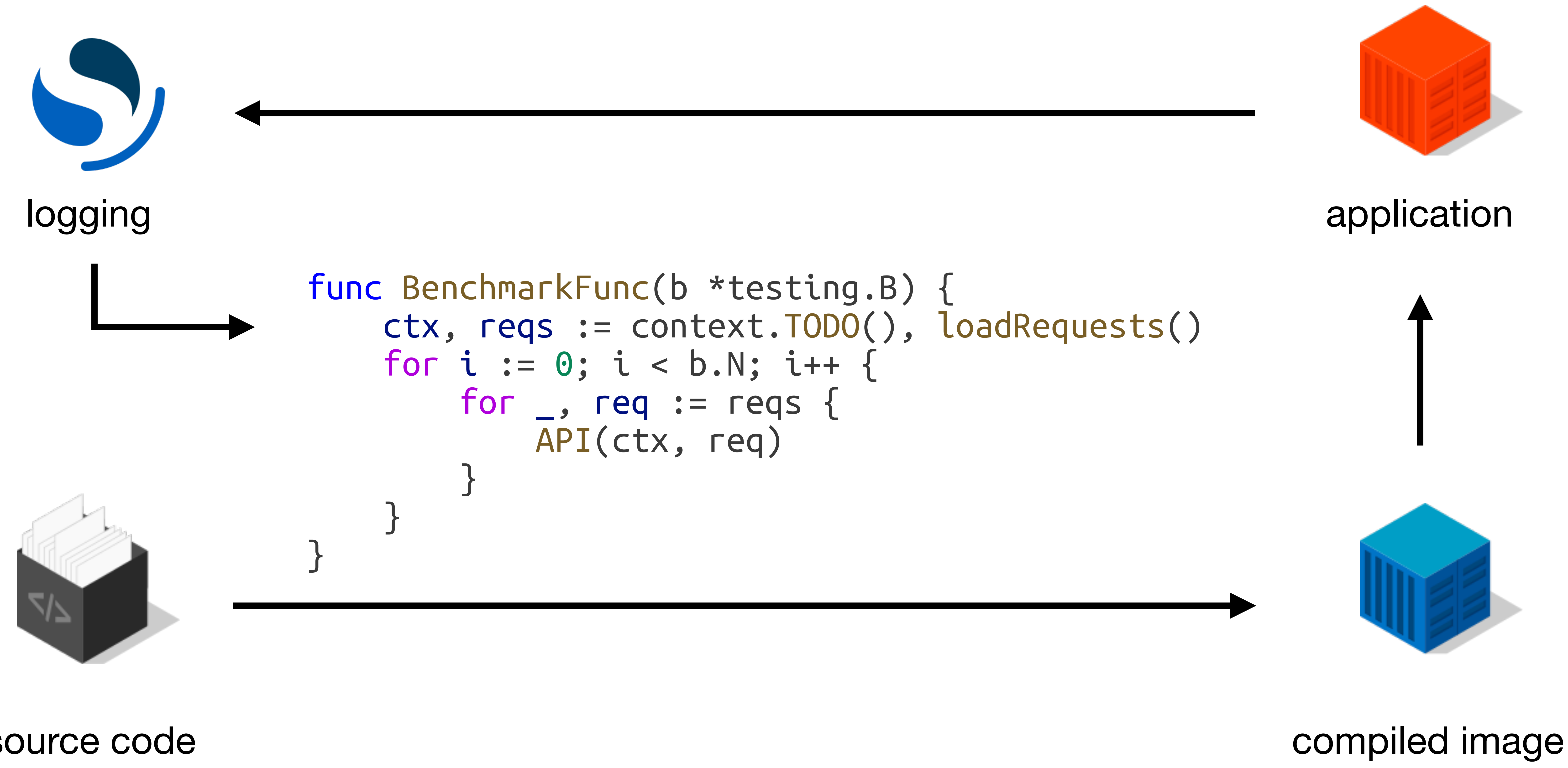
- There are two different approaches:
  - **Traffic Simulation**
  - Double Release



PGO Release Workflow: Traffic Simulation

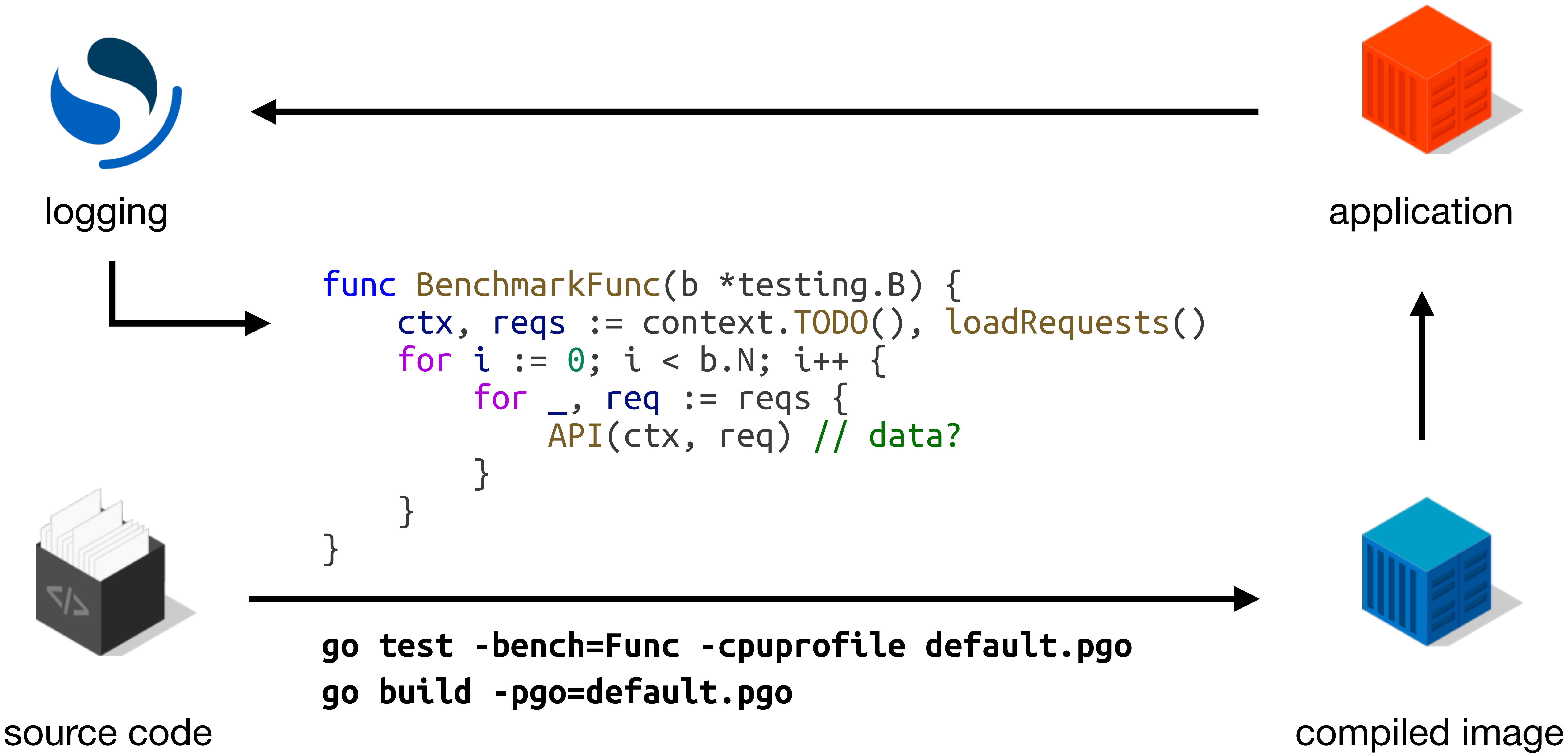


PGO Release Workflow: Traffic Simulation




```go
func BenchmarkFunc(b *testing.B) {
	ctx, reqs := context.TODO(), loadRequests()
	for i := 0; i < b.N; i++ {
		for _, req := range reqs {
			API(ctx, req)
		}
	}
}
```

PGO Release Workflow: Traffic Simulation

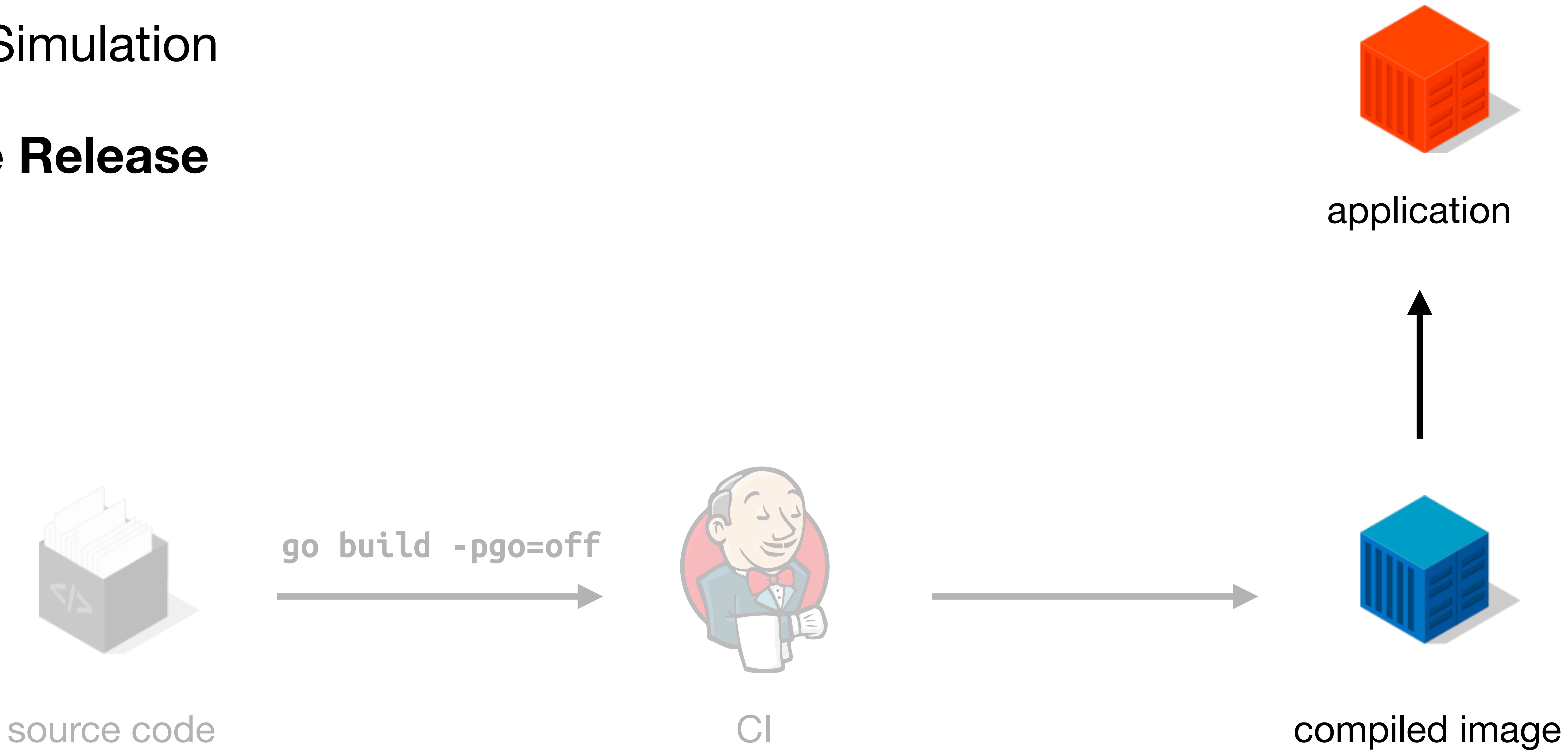



```go
func BenchmarkFunc(b *testing.B) {
    ctx, reqs := context.TODO(), loadRequests()
    for i := 0; i < b.N; i++ {
        for _, req := range reqs {
            API(ctx, req) // data?
        }
    }
}
```

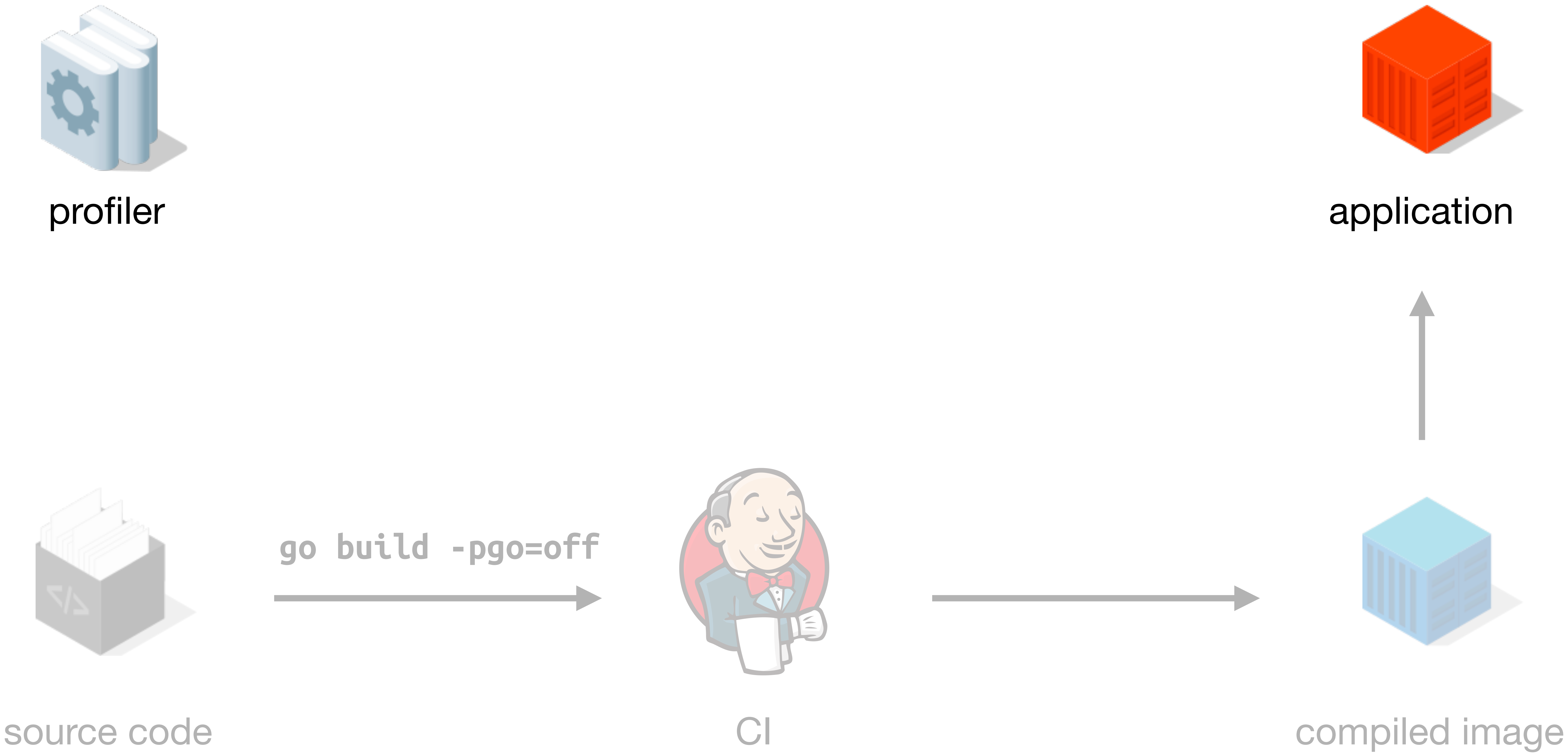
```
go test -bench=Func -cpuprofile default.pgo
go build -pgo=default.pgo
```

PGO Release Workflow

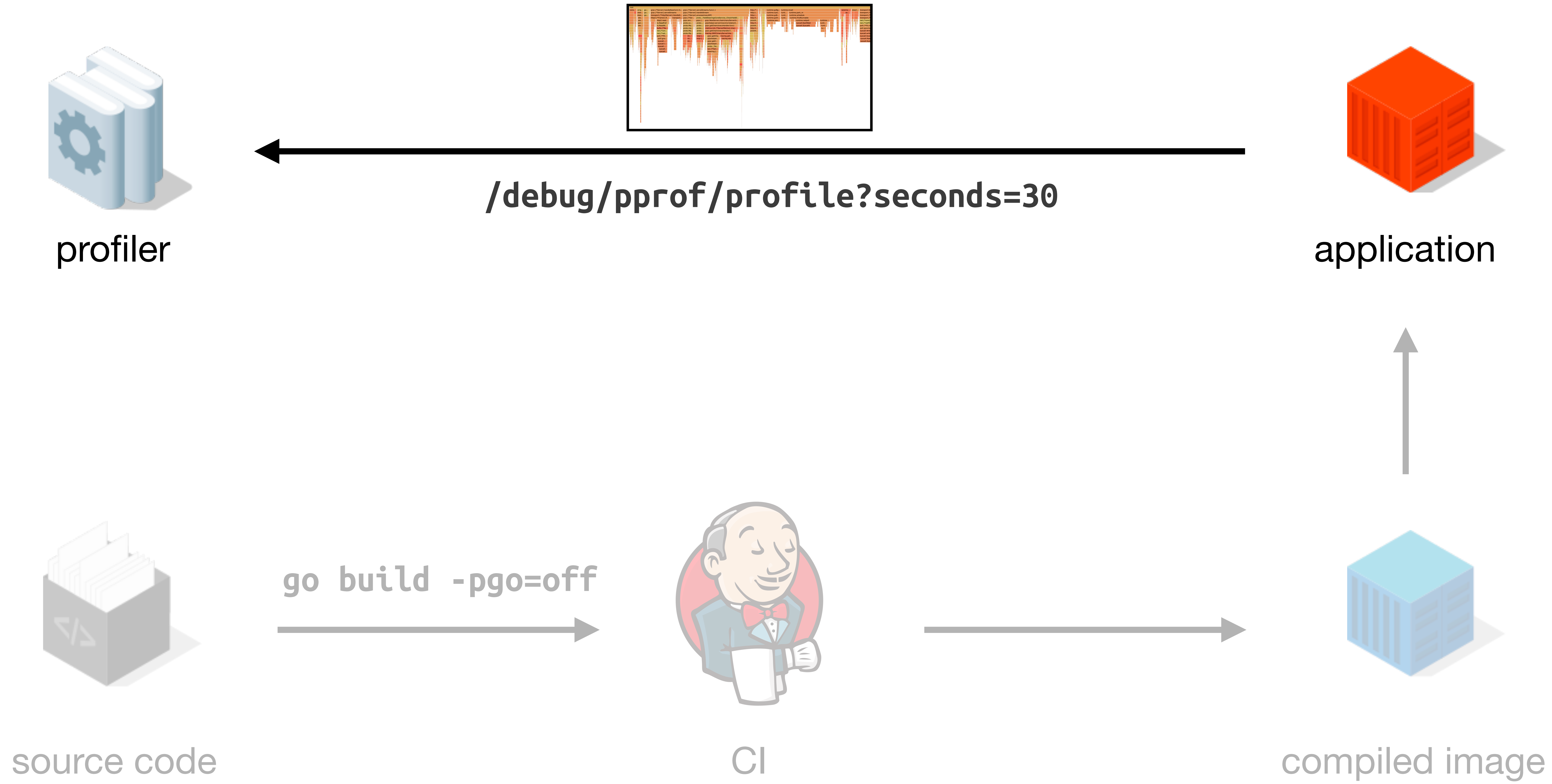
- There are two different approaches:
  - Traffic Simulation
  - **Double Release**



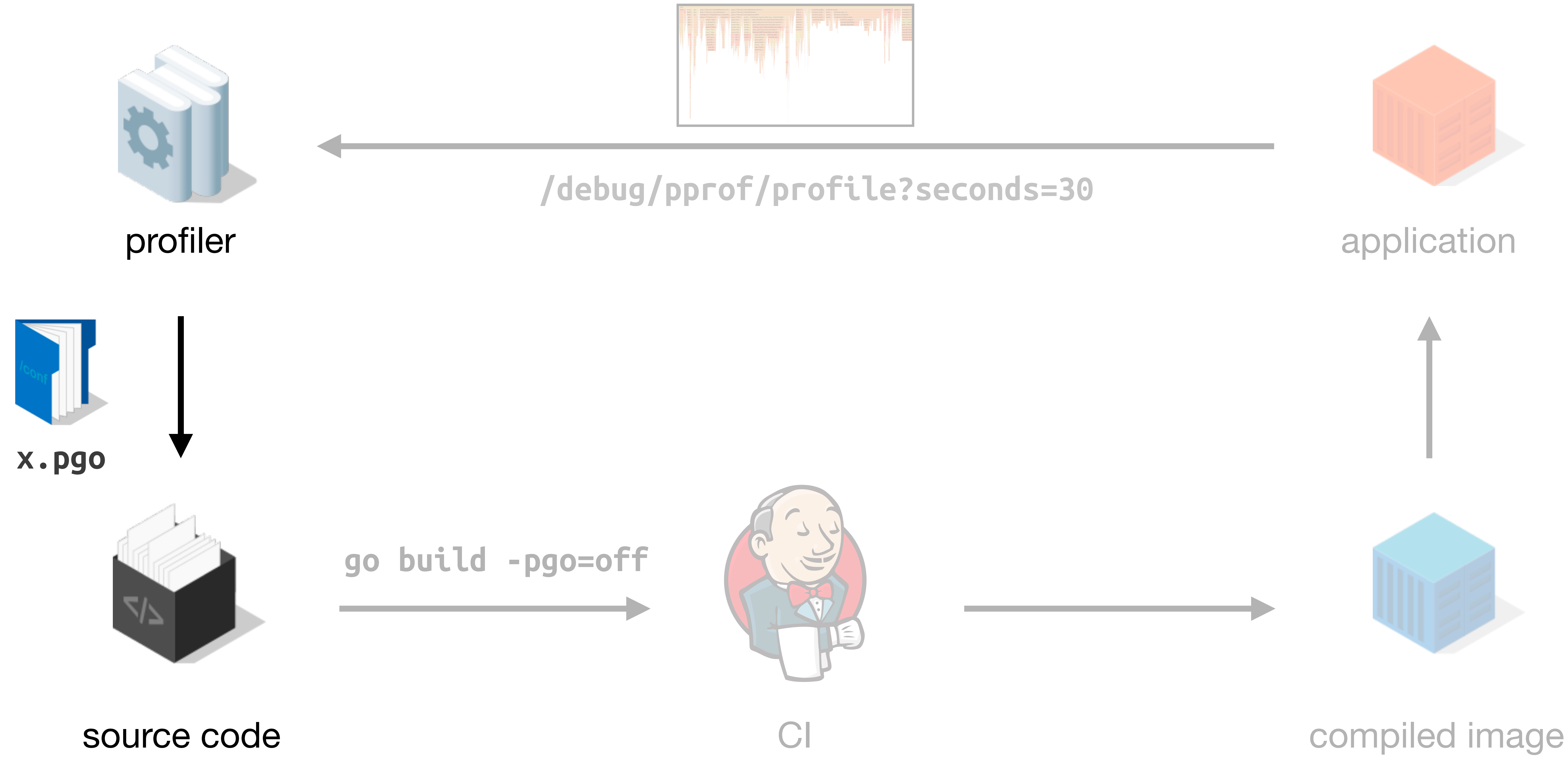
PGO Release Workflow: Feedback Loop



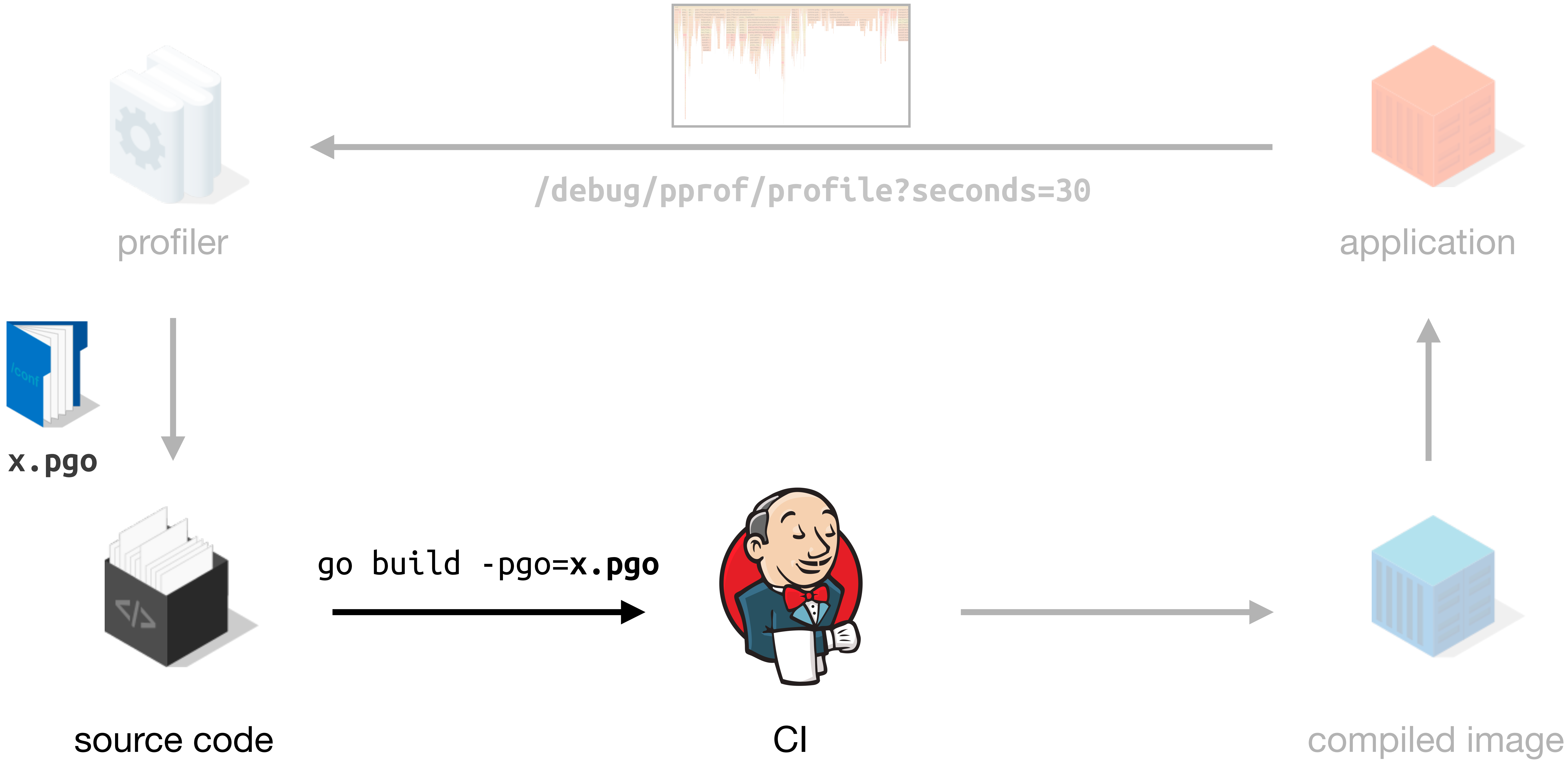
PGO Release Workflow: Feedback Loop



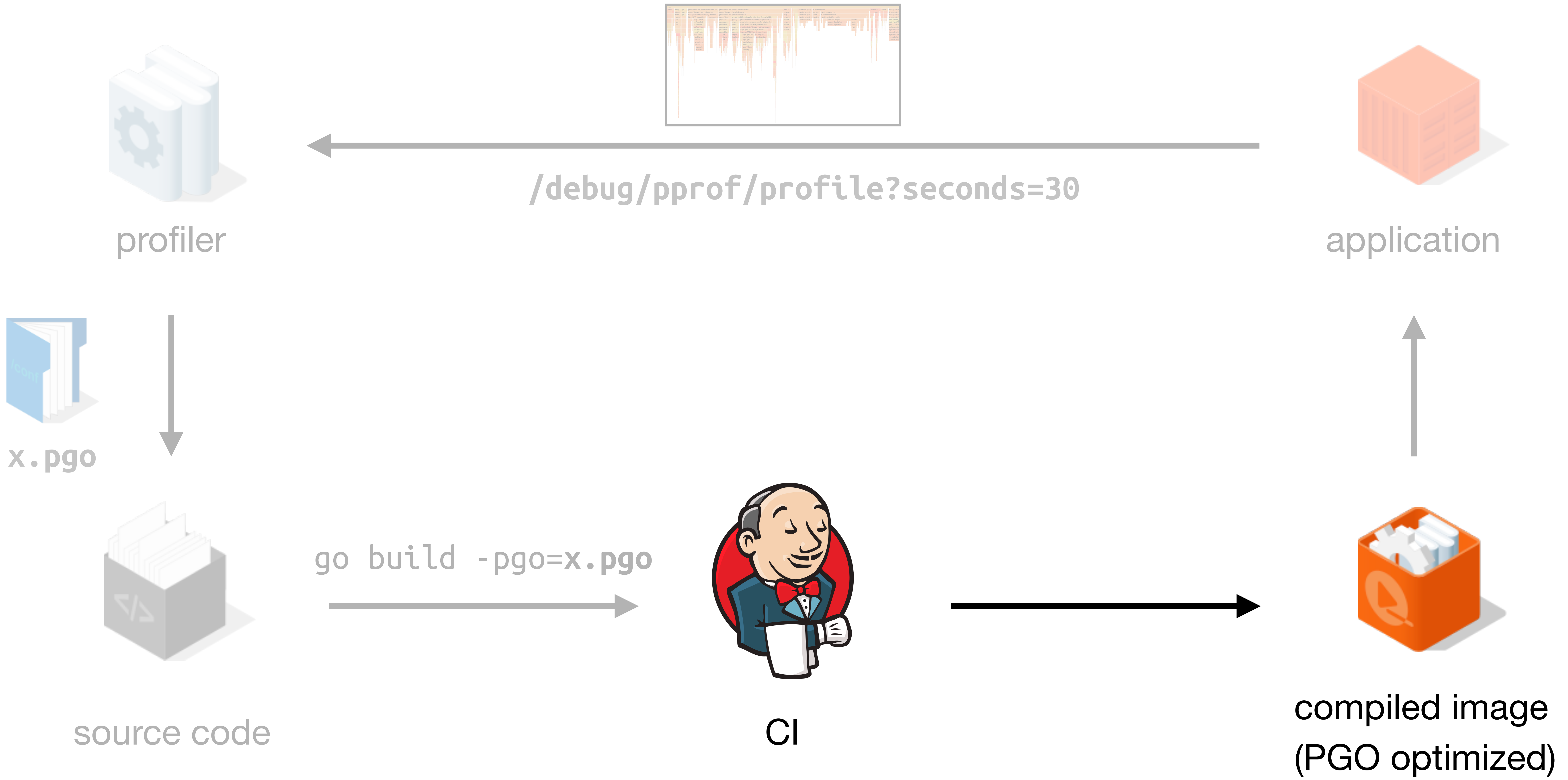
PGO Release Workflow: Feedback Loop



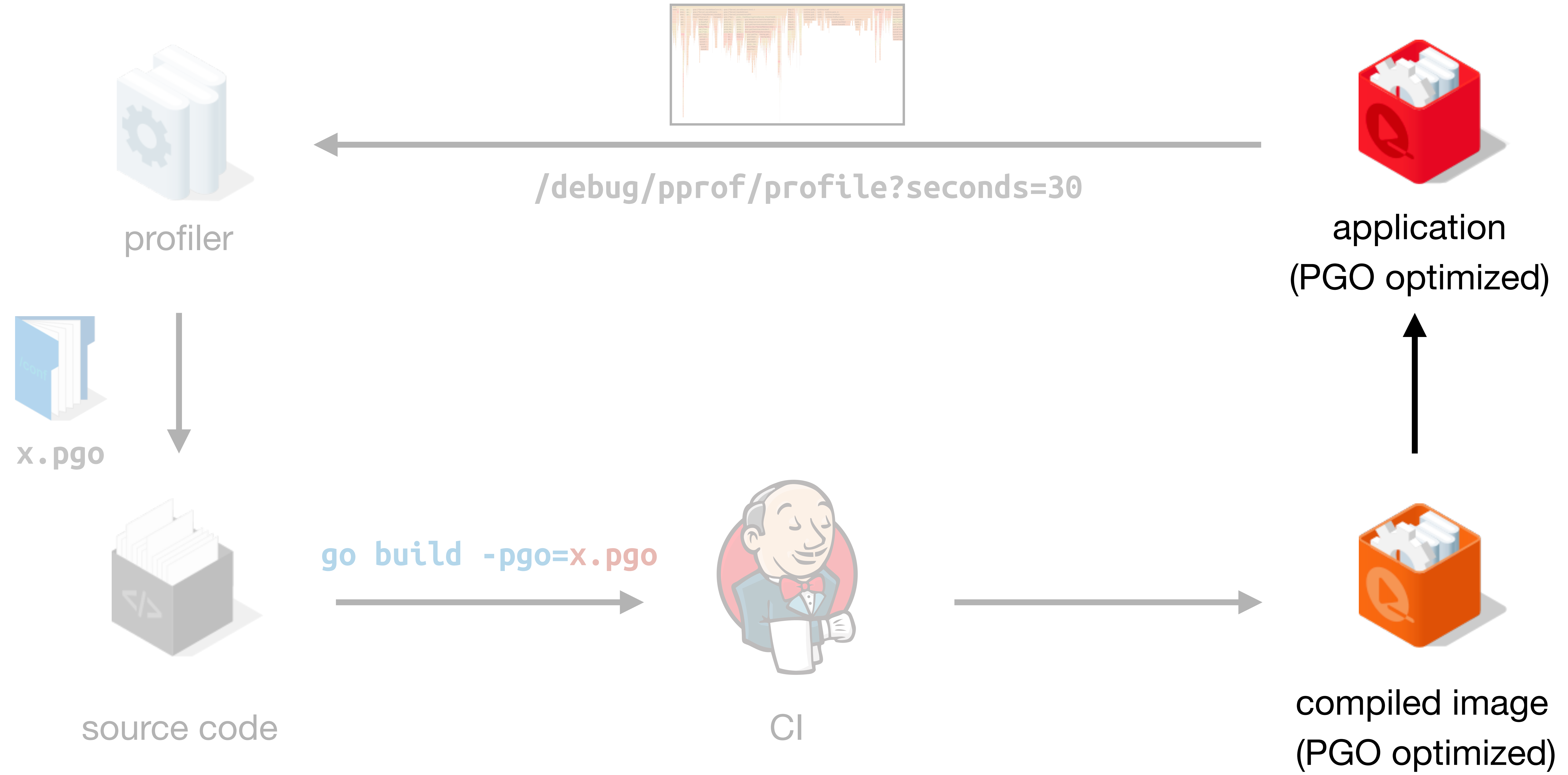
PGO Release Workflow: Feedback Loop



PGO Release Workflow: Feedback Loop



PGO Release Workflow: Feedback Loop



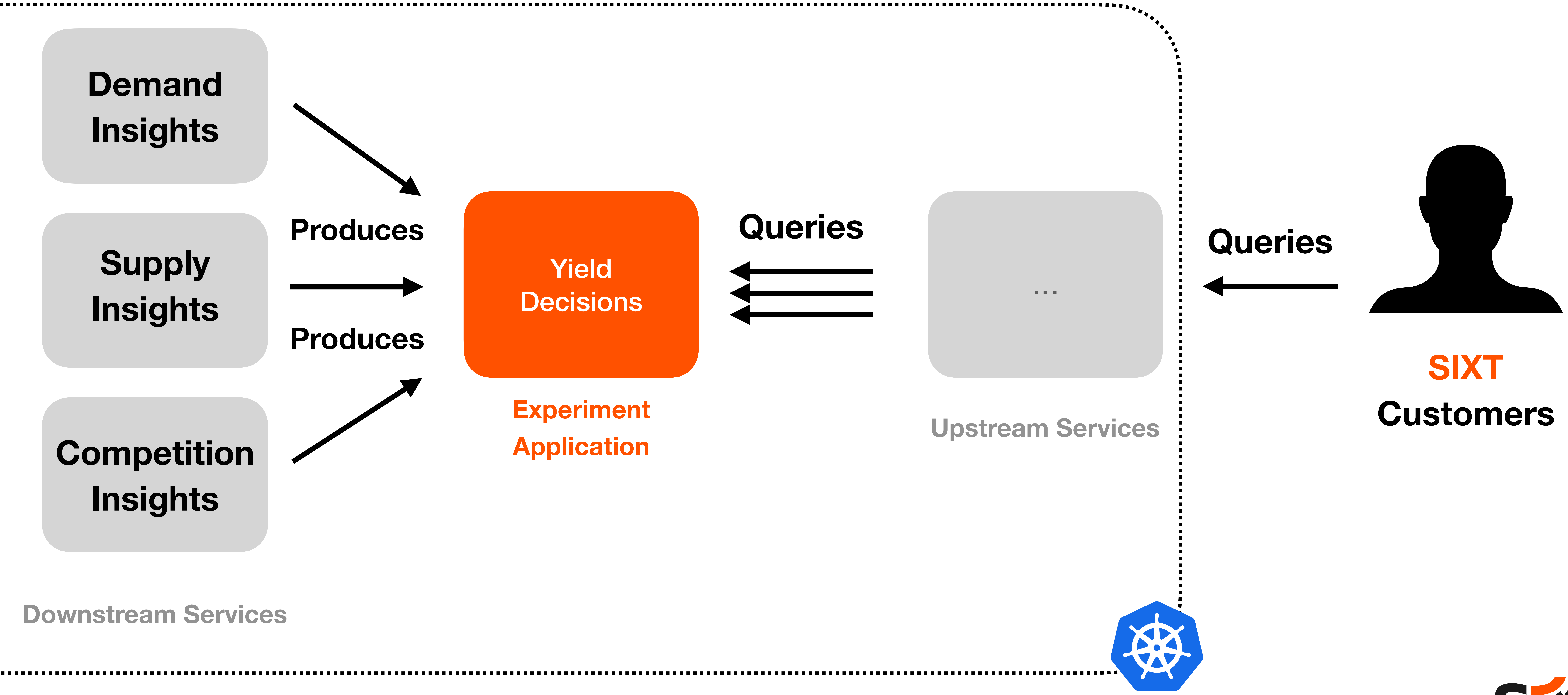
PGO Release Workflow

There are two different approaches:

- **Traffic Simulation**
    - Pros: Do not require deploy 2 times
    - Cons: 1) Simulate production env is non-trivial; 2) Only profiling in smaller scope
- **Double Release**
    - Pros: 1) Fit into PGO's design; 2) Profiling history for inspections
    - Cons: Complicated infrastructure required

Case Study: Production Setup

Case Study: Production Setup



Case Study: Benchmark by Simulation

```
$ go test -pgo=off -v -run=none -bench=BenchmarkCheckYieldCondition -count=10 -cpuprofile without.pgo | tee without-pgo.txt
goos: darwin
goarch: arm64
pkg: internal/steering
BenchmarkCheckYieldCondition
BenchmarkCheckYieldCondition-8        270      5111763 ns/op    356484 B/op      6046 allocs/op
BenchmarkCheckYieldCondition-8        358      4471476 ns/op    365618 B/op      6326 allocs/op
BenchmarkCheckYieldCondition-8        362      3392595 ns/op    366421 B/op      6334 allocs/op
...
```

Case Study: Benchmark by Simulation

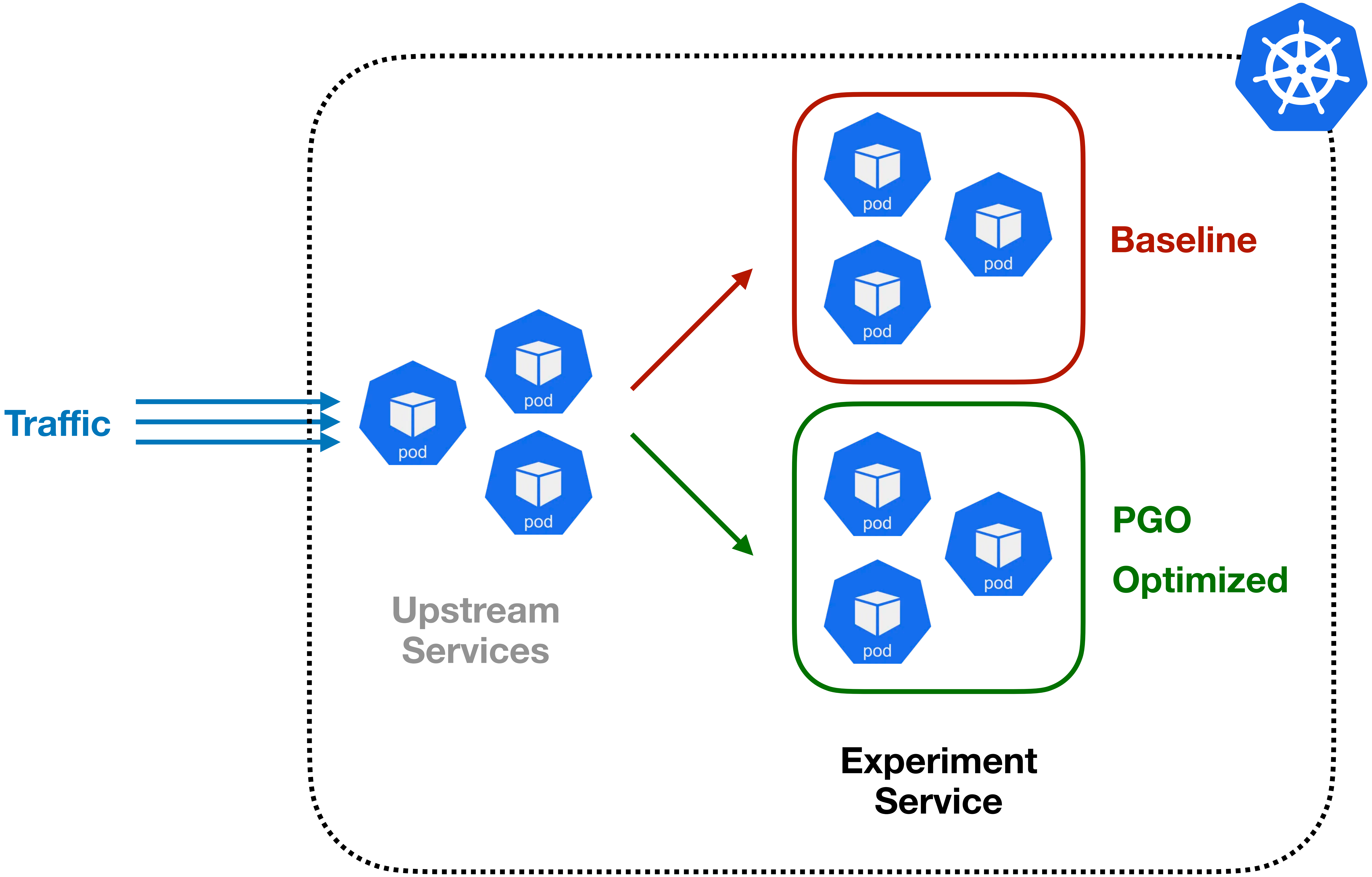
```
$ go test -pgo=off -v -run=none -bench=BenchmarkCheckYieldCondition -count=10 -cpuprofile without.pgo | tee without-pgo.txt
goos: darwin
goarch: arm64
pkg: internal/steering
BenchmarkCheckYieldCondition
BenchmarkCheckYieldCondition-8        270     5111763 ns/op    356484 B/op      6046 allocs/op
BenchmarkCheckYieldCondition-8        358     4471476 ns/op    365618 B/op      6326 allocs/op
BenchmarkCheckYieldCondition-8        362     3392595 ns/op    366421 B/op      6334 allocs/op
...

$ go test -pgo=without.pgo -v -run=none -bench=BenchmarkCheckYieldCondition -count=10 -cpuprofile with.pgo | tee with-pgo.txt
goos: darwin
goarch: arm64
pkg: internal/steering
BenchmarkCheckYieldCondition
BenchmarkCheckYieldCondition-8        355     2823859 ns/op    367670 B/op      6323 allocs/op
BenchmarkCheckYieldCondition-8        409     2463564 ns/op    362959 B/op      6360 allocs/op
BenchmarkCheckYieldCondition-8        450     2505435 ns/op    359412 B/op      6286 allocs/op
...
```

Case Study: Benchmark by Simulation

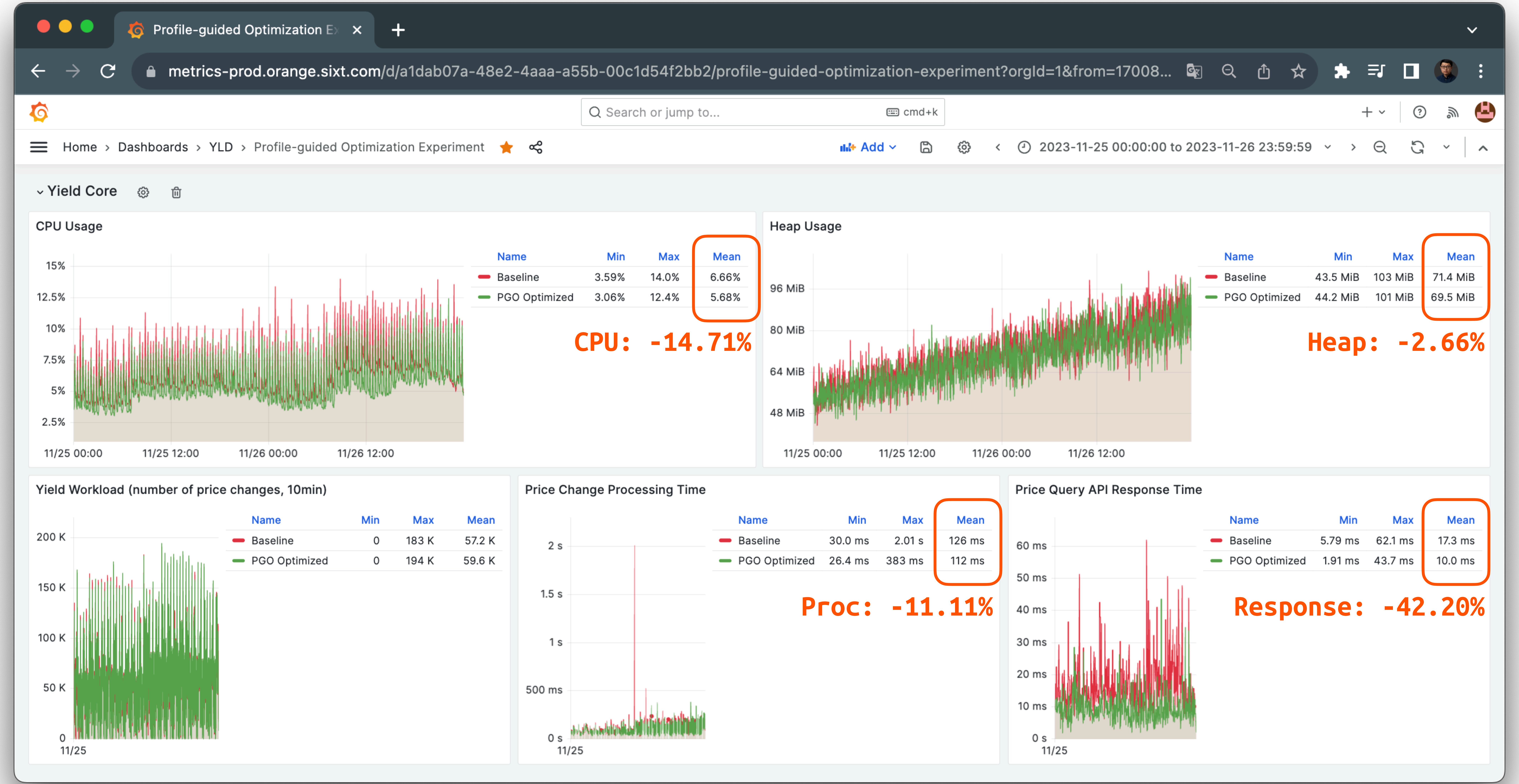
```
$ go test -pgo=off -v -run=none -bench=BenchmarkCheckYieldCondition -count=10 -cpuprofile without.pgo | tee without-pgo.txt
goos: darwin
goarch: arm64
pkg: internal/steering
BenchmarkCheckYieldCondition
BenchmarkCheckYieldCondition-8        270      5111763 ns/op    356484 B/op       6046 allocs/op
BenchmarkCheckYieldCondition-8        358      4471476 ns/op    365618 B/op       6326 allocs/op
BenchmarkCheckYieldCondition-8        362      3392595 ns/op    366421 B/op       6334 allocs/op
...

$ go test -pgo=without.pgo -v -run=none -bench=BenchmarkCheckYieldCondition -count=10 -cpuprofile with.pgo | tee with-pgo.txt
goos: darwin
goarch: arm64
pkg: internal/steering
BenchmarkCheckYieldCondition
BenchmarkCheckYieldCondition-8        355      2823859 ns/op    367670 B/op       6323 allocs/op
BenchmarkCheckYieldCondition-8        409      2463564 ns/op    362959 B/op       6360 allocs/op
BenchmarkCheckYieldCondition-8        450      2505435 ns/op    359412 B/op       6286 allocs/op
...

$ benchstat without-pgo.txt with-pgo.txt
name                   old time/op    new time/op    delta
CheckYieldCondition-8    3.15ms ±42%    2.65ms ±11%  -15.95%  (p=0.000 n=19+19)

name                   old alloc/op   new alloc/op   delta
CheckYieldCondition-8     367kB ± 1%     362kB ± 1%   -1.19%  (p=0.000 n=19+20)

name                   old allocs/op  new allocs/op  delta
CheckYieldCondition-8     6.35k ± 1%     6.34k ± 1%     ~     (p=0.723 n=19+20)
```

Case Study: Production Setup

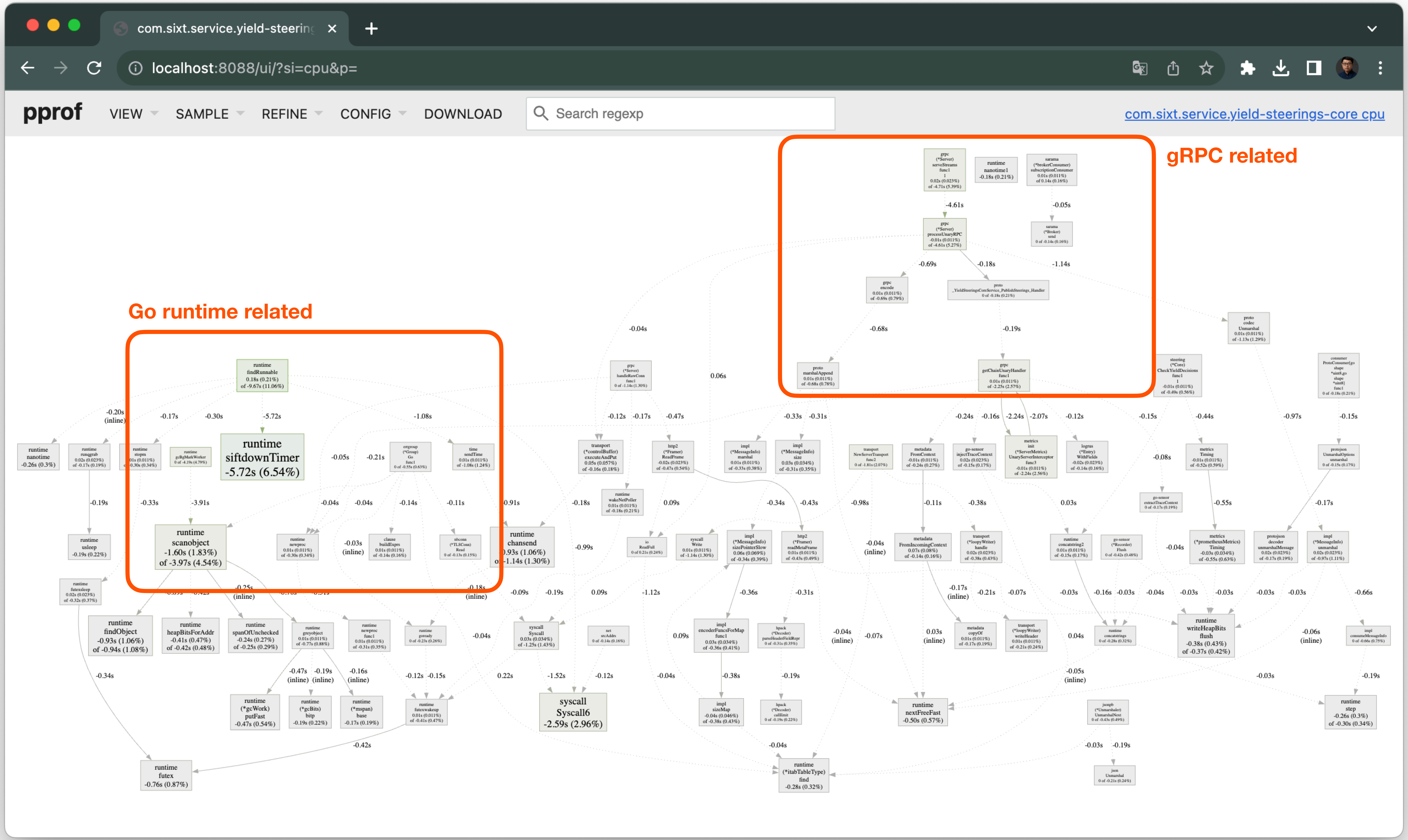
- Canary deployment
    - 5 Pods baseline
    - 5 Pods PGO optimized
- Balanced traffic
    - Each 400 read/s + 5000 write/s



Case Study: Observations

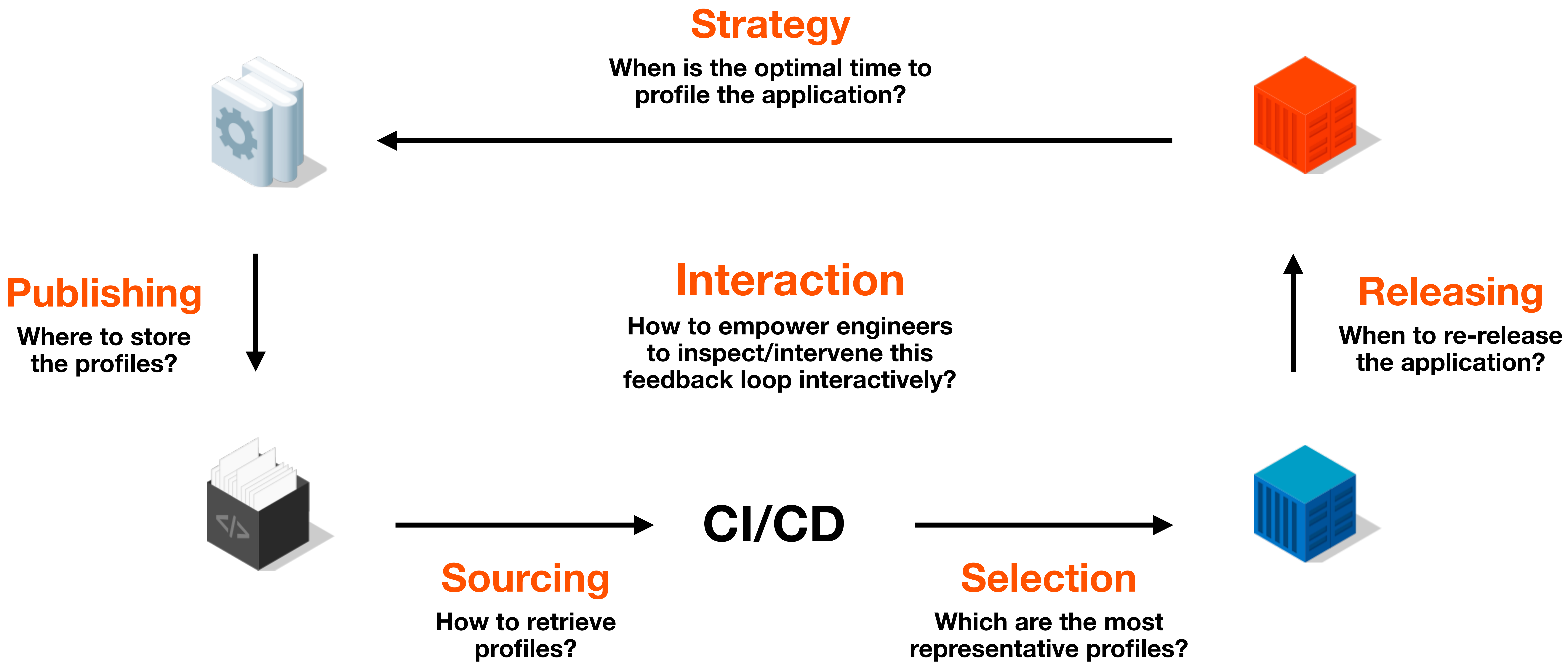


Case Study: Profiling



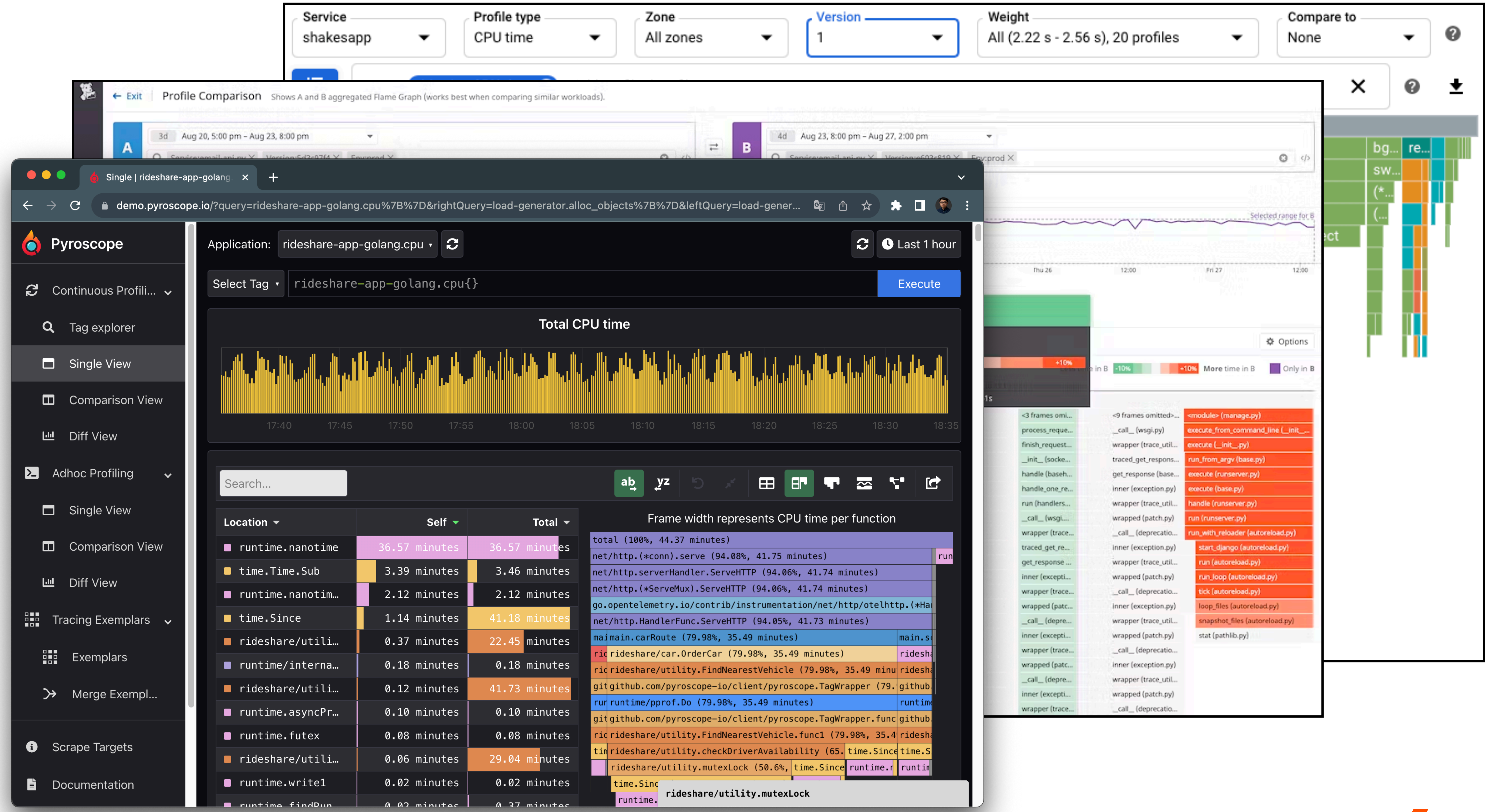
PGO Opportunities in Automation

- There are many opportunities to build solutions to automate PGO release pipeline:



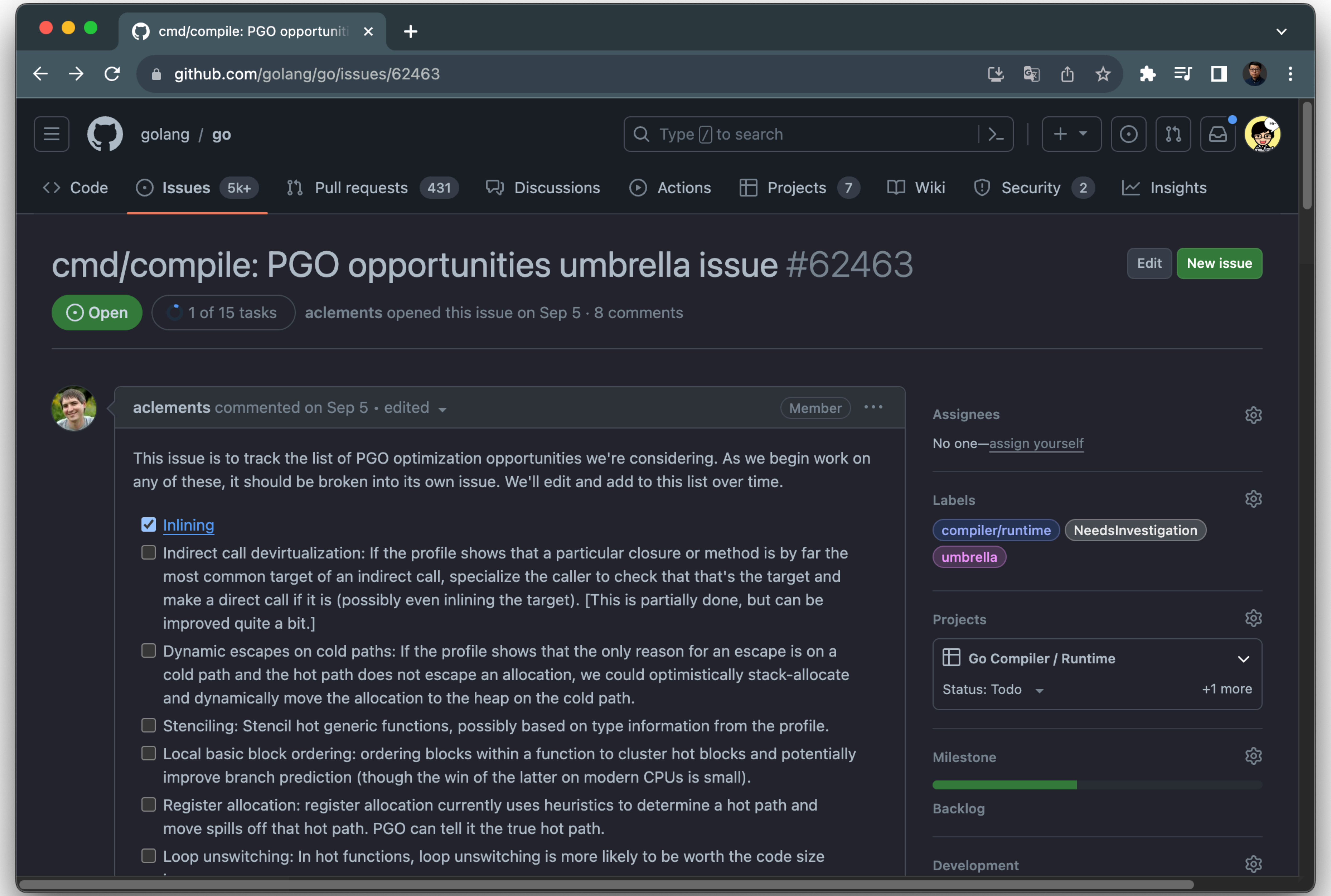
Continuous Profiler Solutions

- Many emerging solutions allow interactive profiling on temporal dimension
  - Pyroscope
  - Datadog
  - Google Cloud Profiler
  - Parca
  - …



PGO Opportunities in Go

- There are many opportunities to contribute to the Go source:
  - Indirect call devirtualization
  - Local basic block ordering
  - Register allocation
  - Function ordering
  - Loop alignment
  - …



Summary and Outlook

Summary

- The idea of data-driven compile time optimization using runtime profiling
- How to use profile-guided optimization in Go application build workflow
- The current status of PGO in Go (inlining and devirtualization)
- The benefits of integrate PGO into CI/CD pipeline
  - Continuous profiling as an infrastructure to support engineers’ daily workflow
  - Without any code changes, our practices and observations on production service showed 5~20% performance improvements and 2~5% memory consumption reduction using PGO

References

- Michael Pratt. "Profile-guided optimization". 2023. https://go.dev/doc/pgo
- Chen, Dehao, David Xinliang Li, and Tipp Moseley. "AutoFDO: Automatic feedback-directed optimization for warehouse-scale applications." Proceedings of the 2016 International Symposium on Code Generation and Optimization. 2016. https://doi.org/10.1145/2854038.2854044
- Changkun Ou. "Go: A Documentary”. 2023. https://golang.design/history/#compiler
- Changkun Ou. "Reliable Benchmarking in Go”. Mar 26, 2020. https://changkun.de/talk/gobench.pdf
- Rotem, Nadav, and Chris Cummins. "Profile guided optimization without profiles: A machine learning approach." arXiv preprint arXiv:2112.14679 (2021).
- Felix Geisendörfer. Go Profiling and Observability from Scratch. GopherCon’ 21. 2021