



Go Channel Slowdown with more CPU

16 August 2024

Grant Stephens

Staff Engineer





Grant Stephens

Logging @ Fastly

Ex mechanic

Recently bought a chainsaw



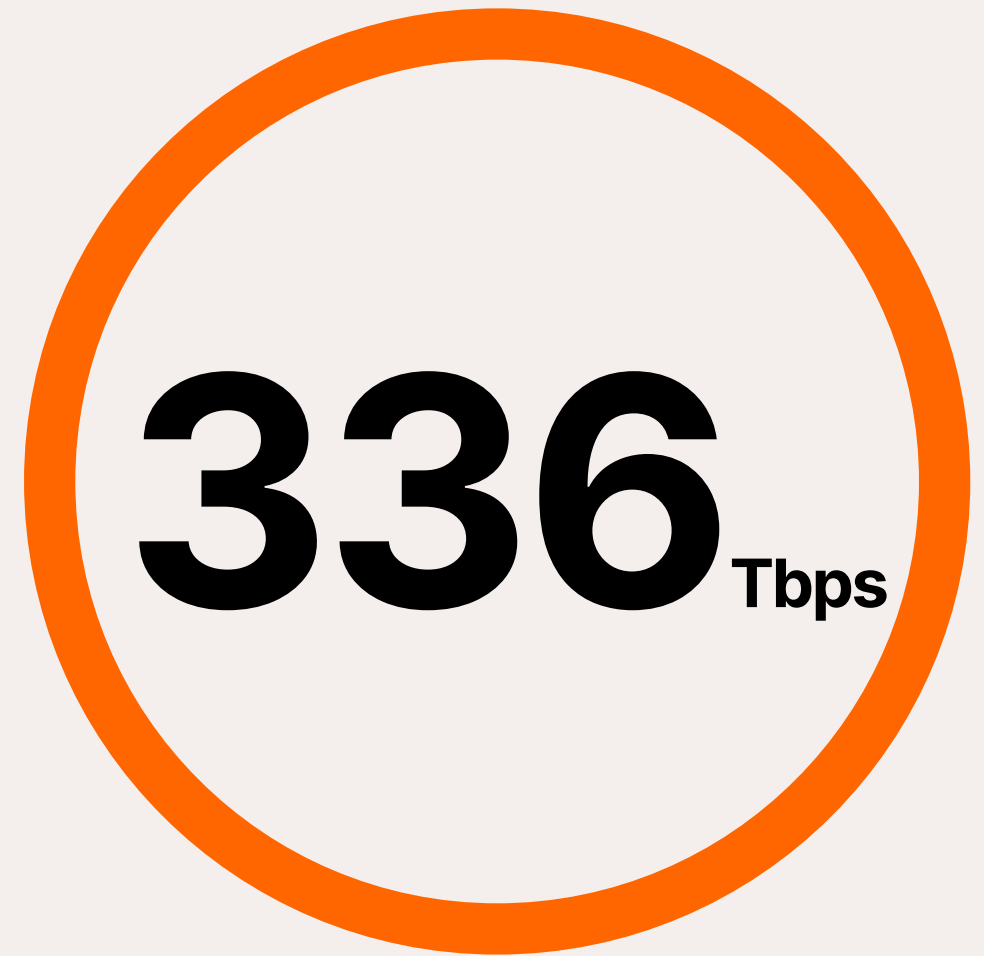
fastly[®]



Edge Cache Nodes



Countries



Edge network capacity

At 31 March 2024

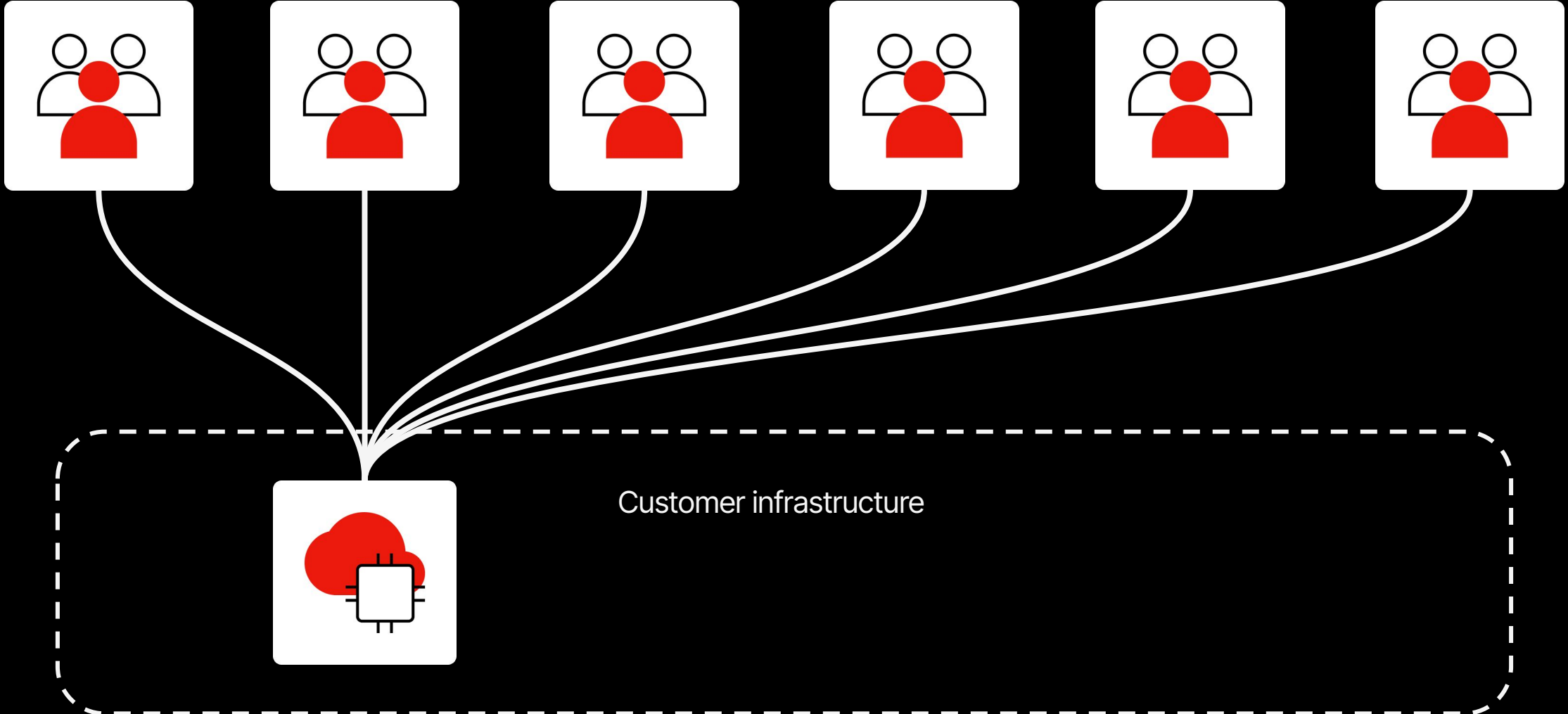


Context

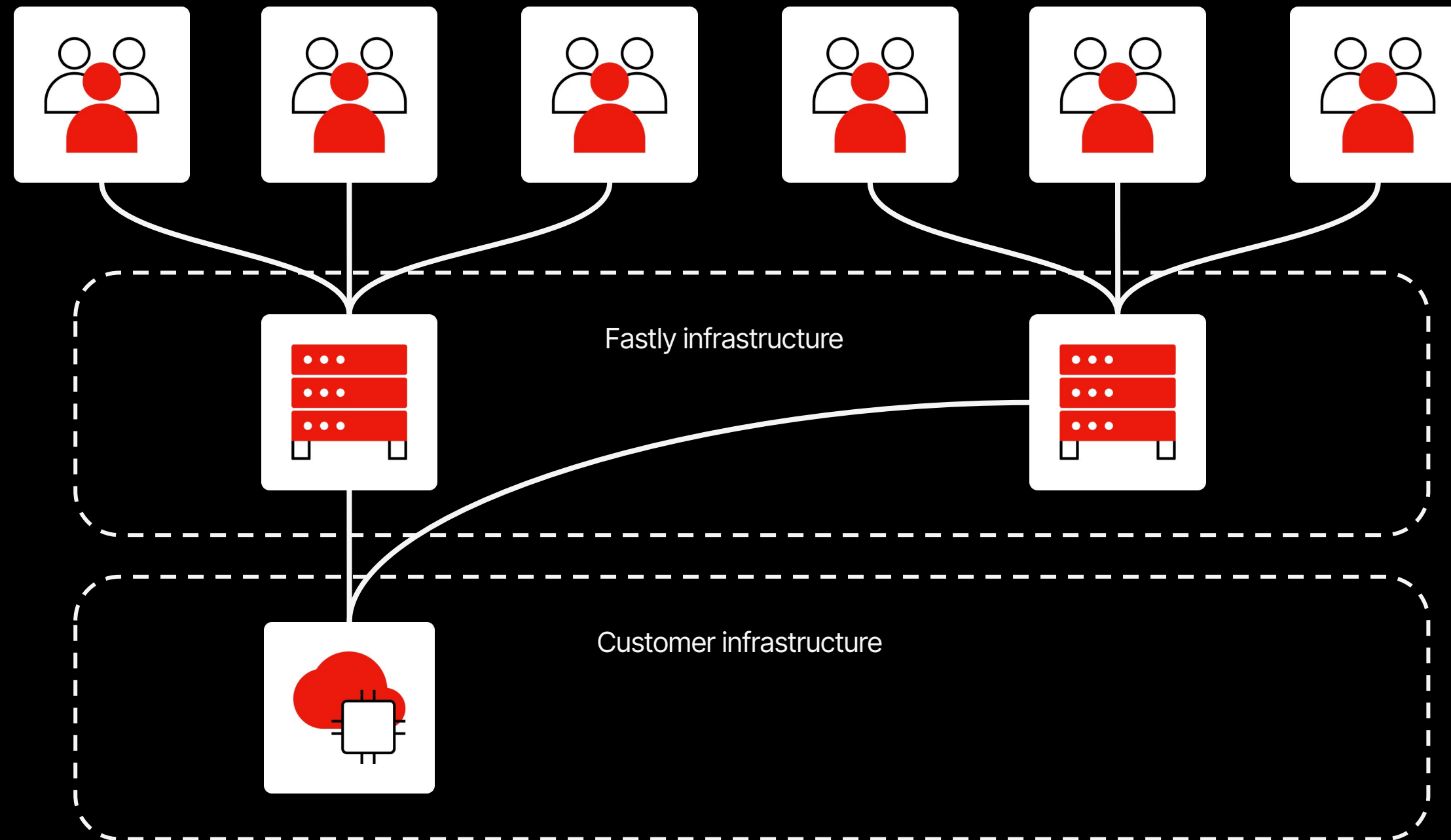


```
ctx.Background()
```

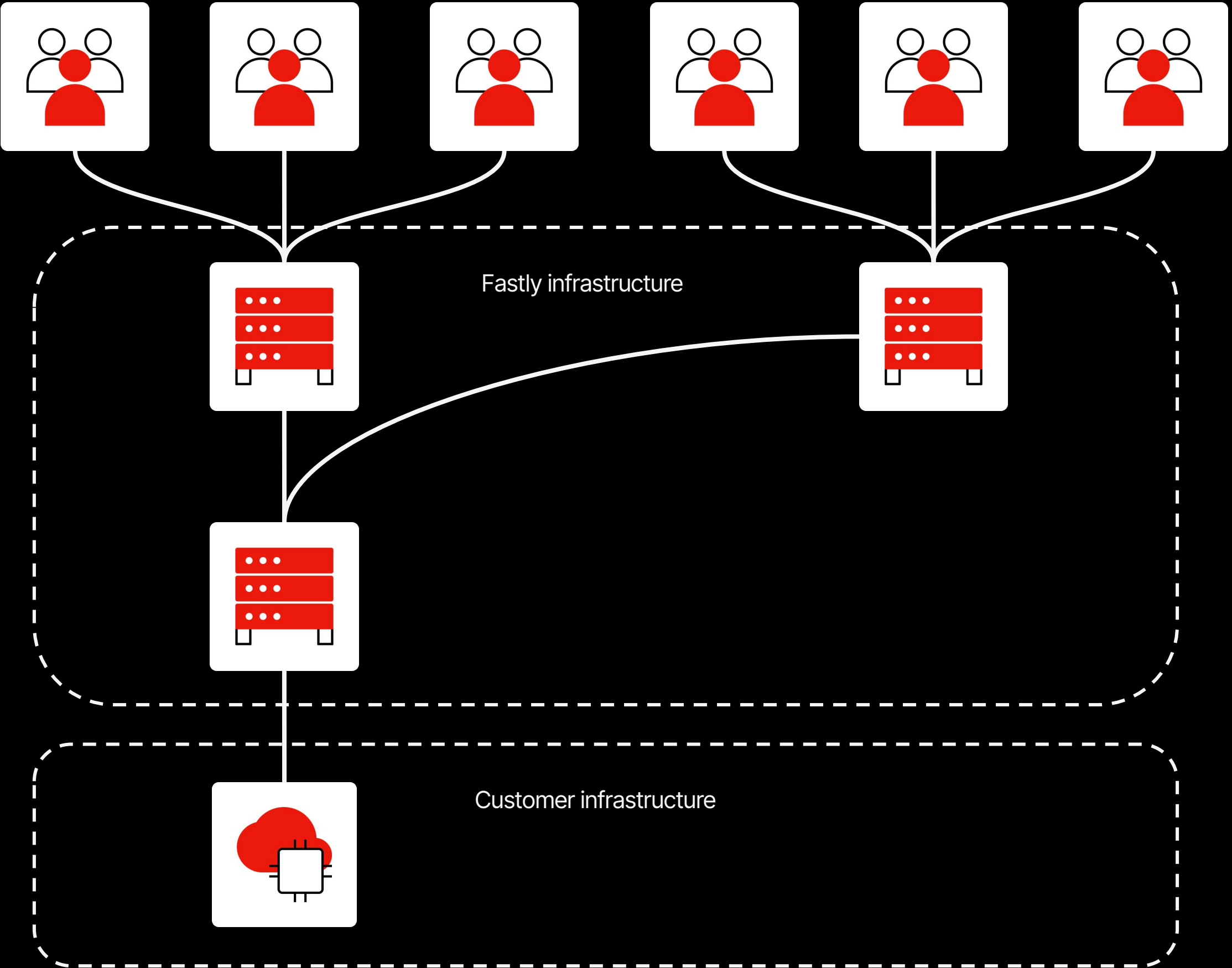
No Edge



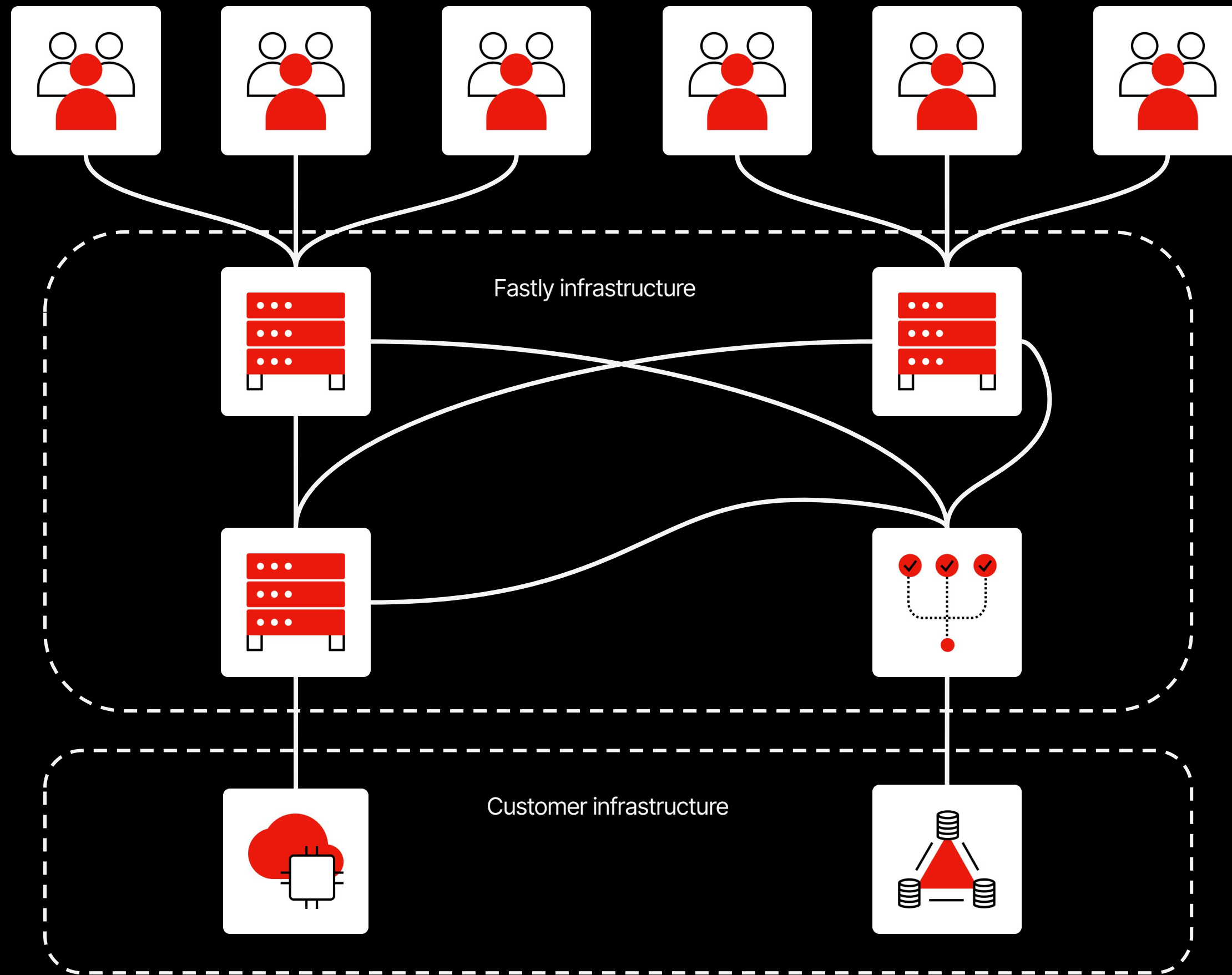
Basics

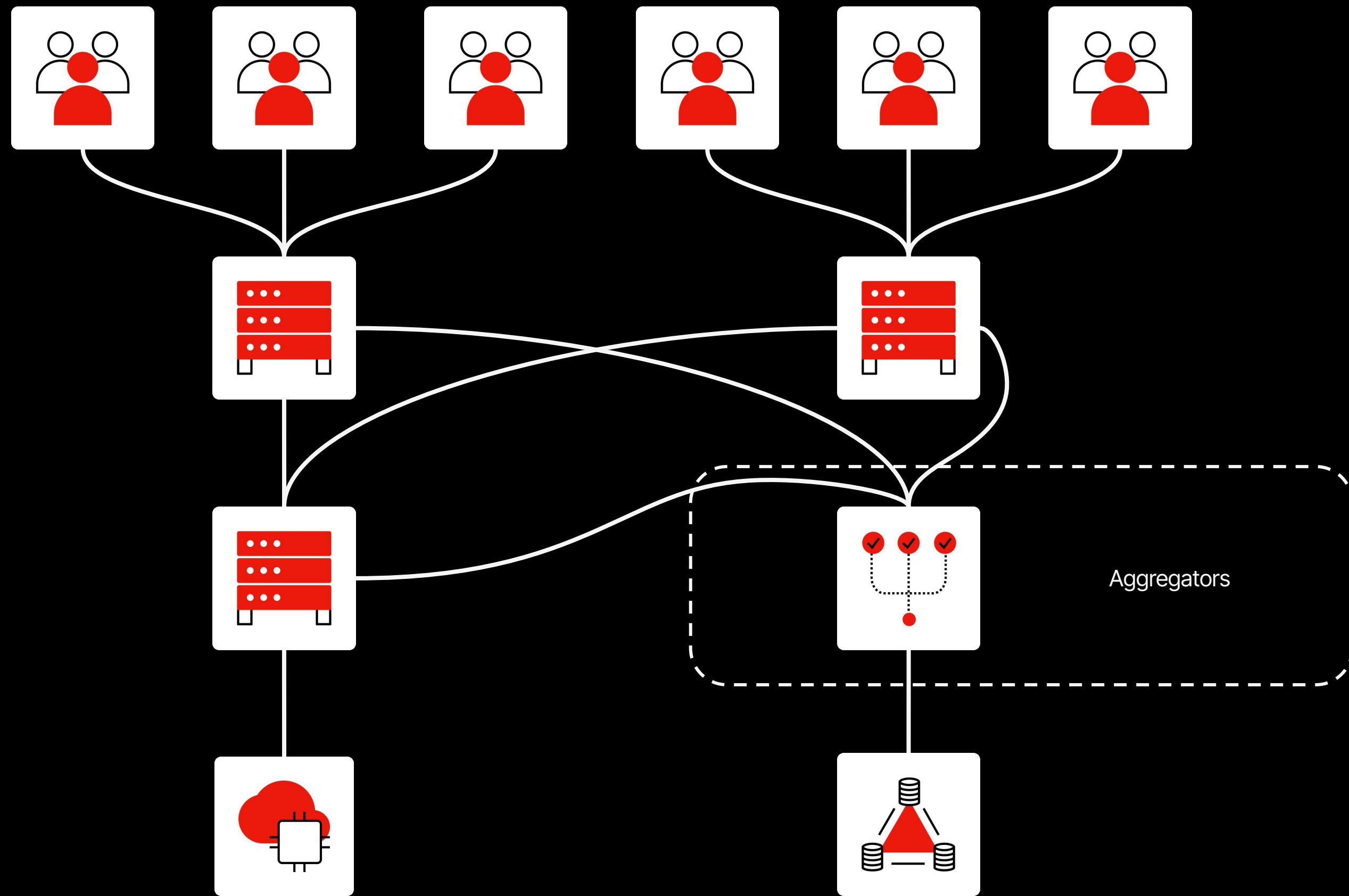


Shielding



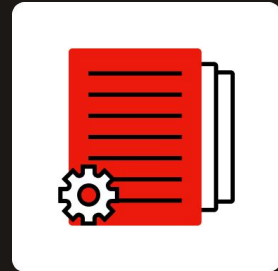
Logging





Some of our customers





36.9_M

Average logs/s delivered
As of 1st August 2024

Clearly
valuable



One day, it got slow...



One day it stopped getting
faster...



Check the logs
(Yes, the logging system logs)

Head Scratching



`/sync/mutex/wait/total:seconds`

Approximate cumulative time goroutines have spent blocked on a `sync.Mutex`, `sync.RWMutex`, or `runtime-internal` lock. This metric is useful for identifying global changes in lock contention. Collect a mutex or block profile using the `runtime/pprof` package for more detailed contention data.

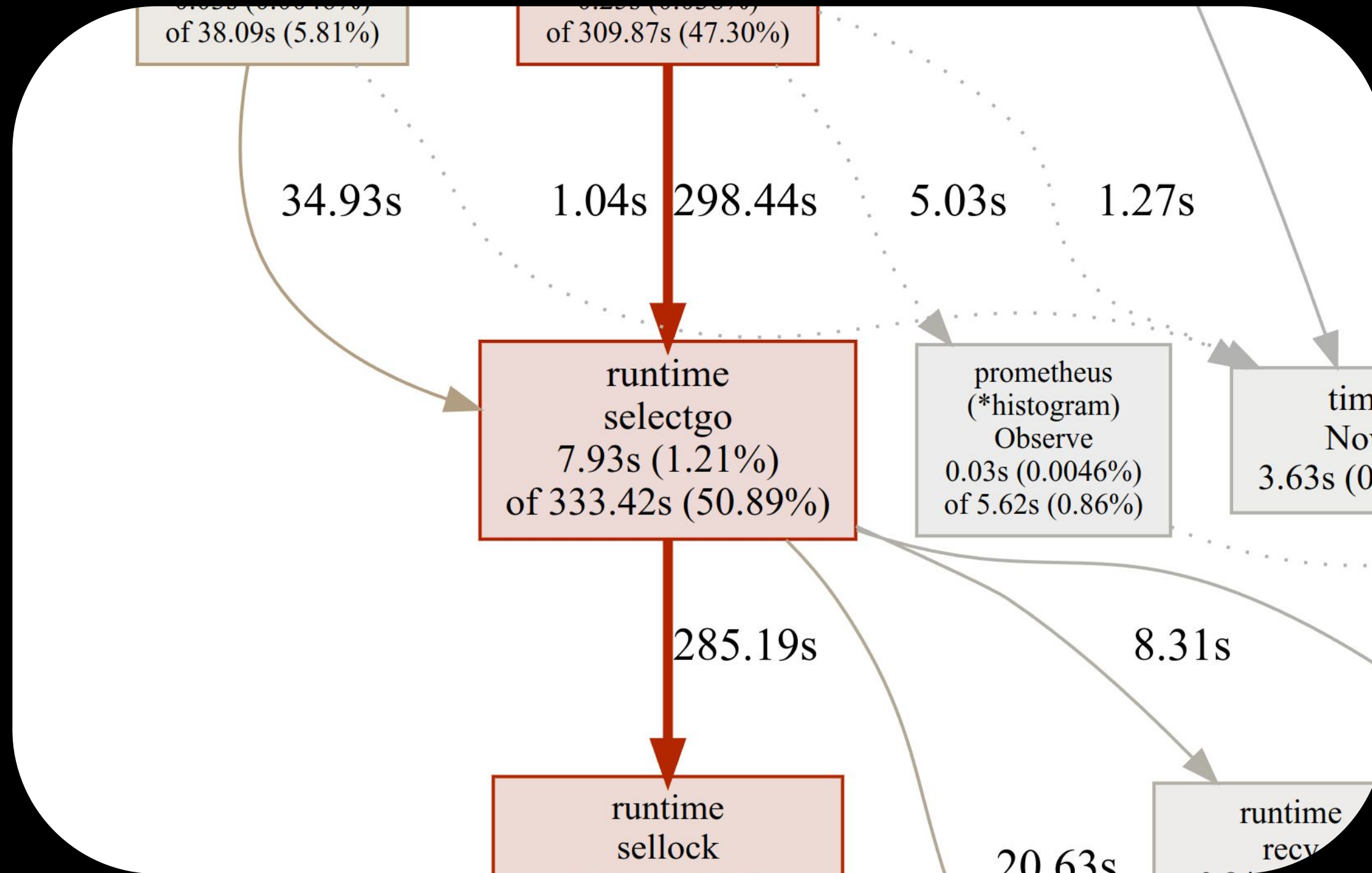


Profile it

Grab a profile

```
ssh -oProxyJump=bastion myBadlyBehavingHost \  
'curl http://127.0.0.1:1310/debug/pprof/profile' > badProfile.prof
```

Hint...



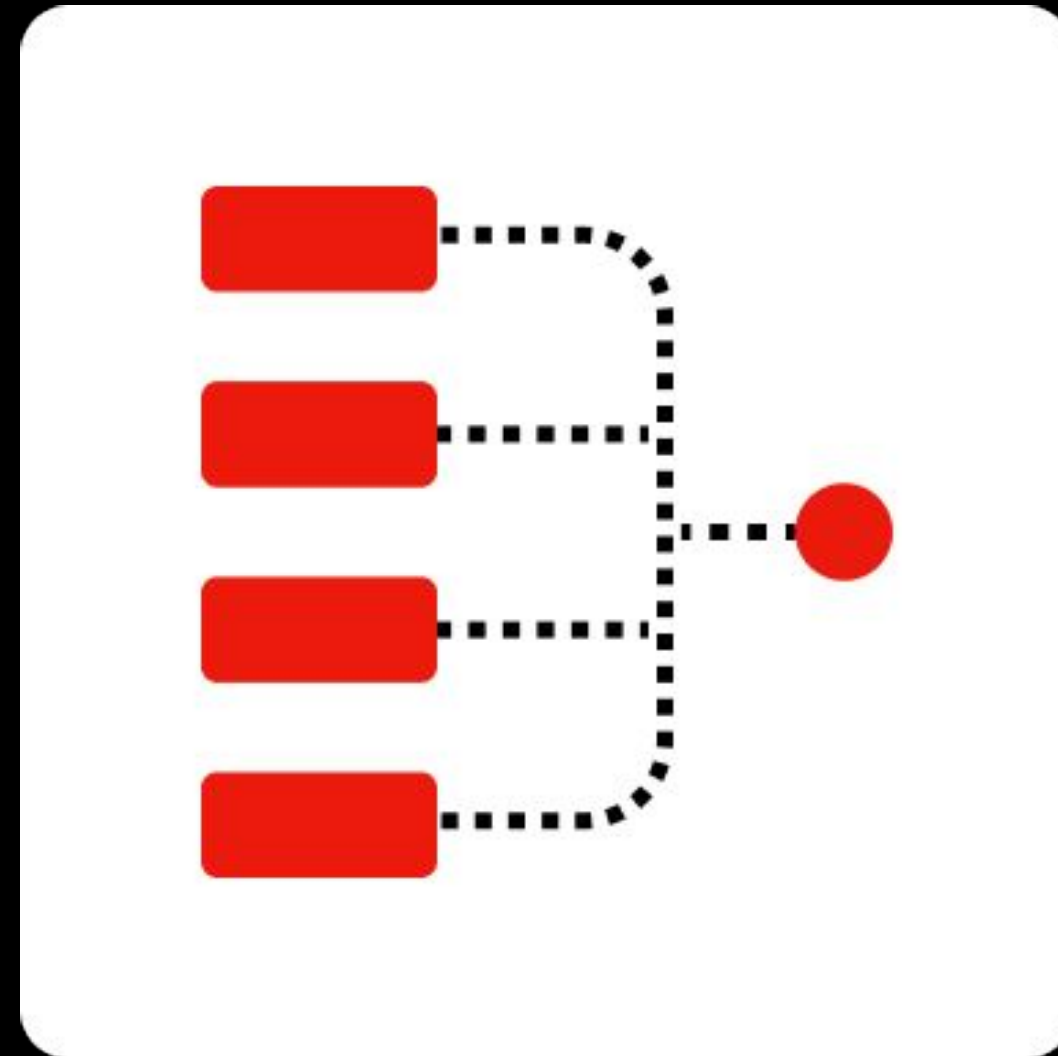


Channels have locks!?!



Channels have locks!!!

Single channel bottleneck





Channels are fast right?

Phil's wisdom

So just how fast are channels anyway?



Phil Pearl · [Follow](#)

Published in Ravelin Tech Blog · 4 min read · Jan 21, 2017



548



7



Basic Benchmark

Send a byte, do nothing with it

```
func BenchmarkChannelOneByte(b *testing.B) {
    ch := make(chan byte, 4096)
    wg := sync.WaitGroup{}
    wg.Add(1)
    go func() {
        defer wg.Done()
        for range ch {
        }
    }()
    b.SetBytes(1)
    b.ReportAllocs()
    b.ResetTimer()
    for i := 0; i < b.N; i++ {
        ch <- byte(i)
    }
    close(ch)
    wg.Wait()
}
```



10M

Channel throughput per second

Fast...

Head Scratching





Benchmark \neq Production



Now... about those aggregators

128

Cores

1TB

Memory

128 ≠ 8

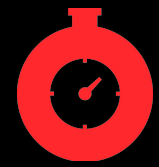


Let's benchmark on production hardware

Basic Benchmark

```
go func() {  
    for range b.ch {  
    }  
}()
```

```
go func() {  
    for {  
        select {  
        case <-b.ch:  
        case <-b.close:  
            return  
        }  
    }  
}()
```



Side quest: Benchmarking Tips and Tricks

Dave Cheney

The acme of foolishness

HIGH PERFORMANCE GO

PRACTICAL GO

INTERNETS OF INTEREST

ABOUT

How to write benchmarks in Go

This post continues a series on the testing package I started a few weeks back. You can read the previous article on [writing table driven tests here](#). You can find the code mentioned below in the <https://github.com/davecheney/fib> repository.

Building Benchmarks

```
go test -bench="BenchmarkChannel" -c .
```

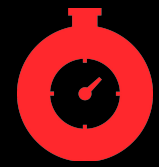
Running Built Benchmarks

```
./chanbench.test -test.bench BenchmarkChannel -test.cpu=1,2.. -test.count=10
```

```
$ benchstat old.txt new.txt
goos: linux
goarch: amd64
pkg: golang.org/x/perf/cmd/benchstat/testdata
```

	old.txt	new.txt	
	sec/op	sec/op	vs base
Encode/format=json-48	1.718μ ± 1%	1.423μ ± 1%	-17.20% (p=0.000 n=10)
Encode/format=gob-48	3.066μ ± 0%	3.070μ ± 2%	~ (p=0.446 n=10)
geomean	2.295μ	2.090μ	-8.94%

Benchstat FTW!



Side quest complete.

Where were we...

Basic Benchmark

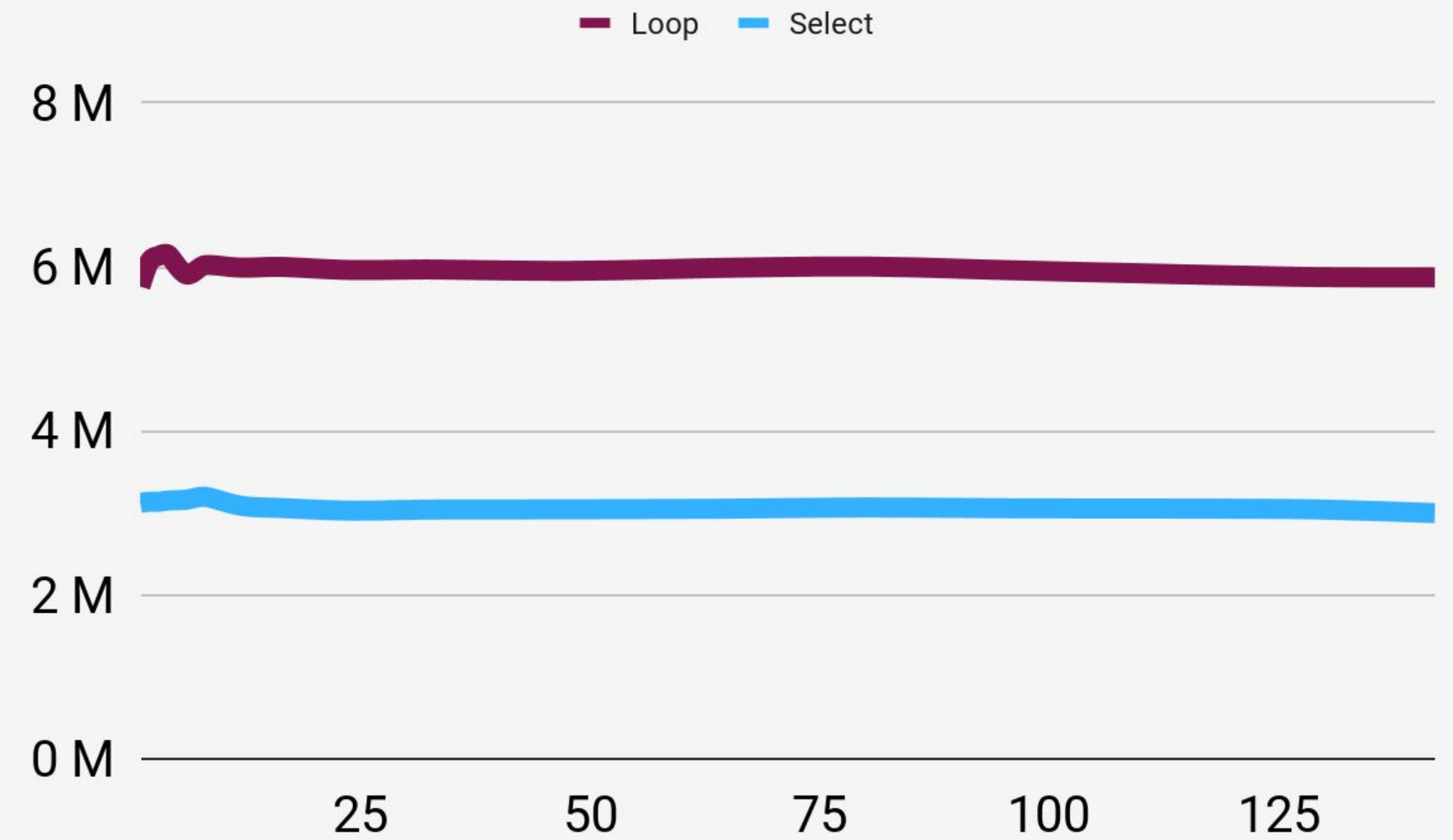
```
go func() {  
    for range b.ch {  
    }  
}()
```

```
go func() {  
    for {  
        select {  
        case <-b.ch:  
        case <-b.close:  
            return  
        }  
    }  
}()
```

Basic Benchmark

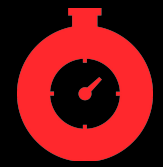
Send some bytes on a channel

Read them off as quickly as possible





Thank you



Try again

Basic Benchmark

```
b.Run(tst.name, func(b *testing.B) {  
    f := tst.new()  
    f.Start()  
    defer f.Stop()  
    b.ResetTimer()  
    for i := 0; i < b.N; i++ {  
        f.Put(msgs[i%len(msgs)])  
    }  
    b.ReportAllocs()  
})
```

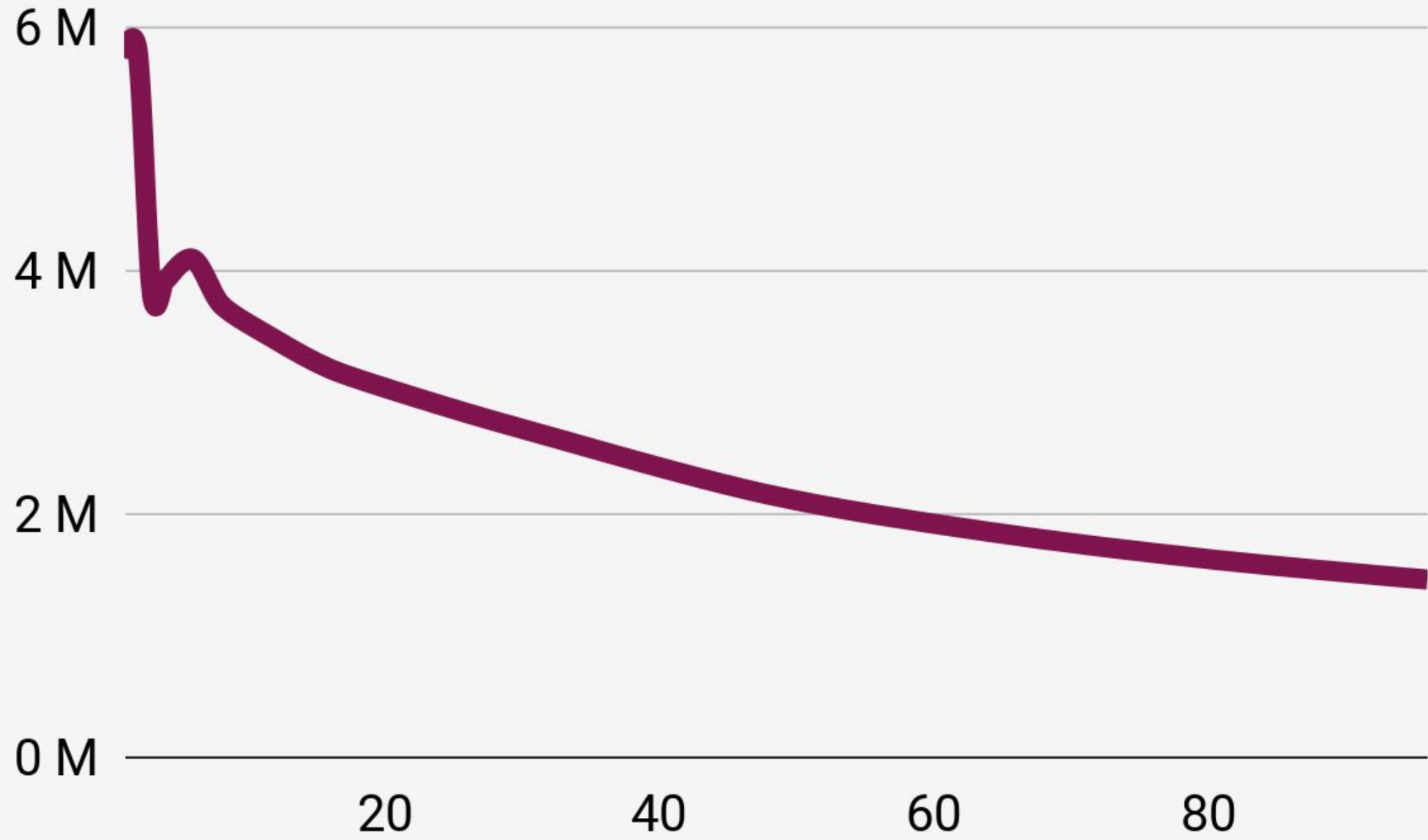
Basic Benchmark.. In Parallel


```
b.Run(tst.name, func(b *testing.B) {
    f := tst.new()
    f.Start()
    b.ResetTimer()
    defer f.Stop()
    i := 0
    b.RunParallel(func(pb *testing.PB) {
        for pb.Next() {
            f.Put(msgs[i%len(msgs)])
            i++
        }
    })
    b.ReportAllocs()
})
```



This slide left intentionally blank
to build suspense...

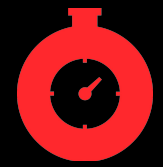
CPU Benchmark





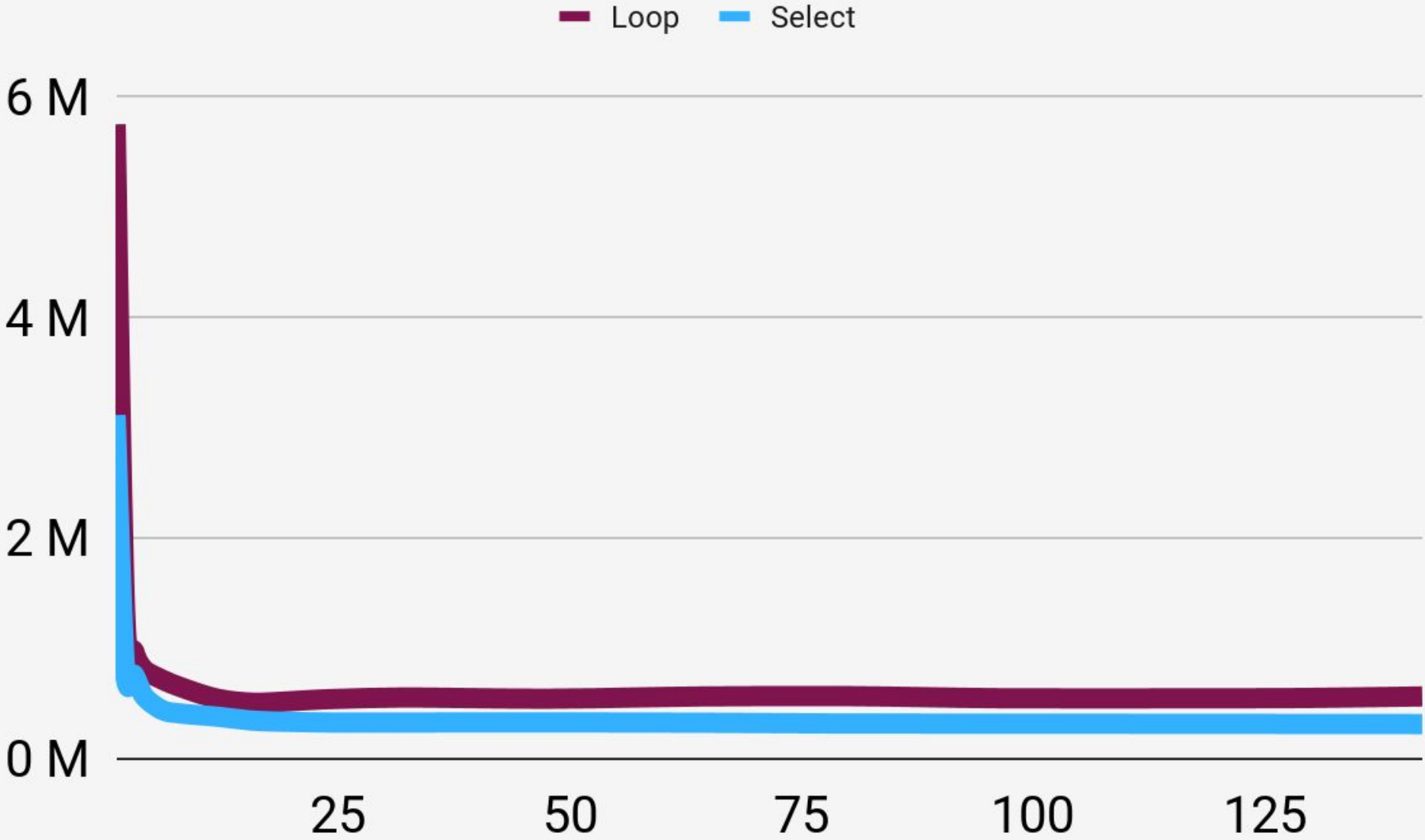
÷3

Slowdown in channel speed from 1 to 128 CPUs

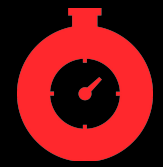


Range and Select

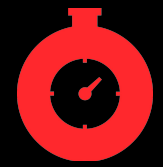
CPU Benchmark



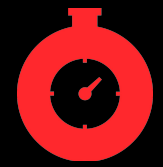
Same effect for loops & select



Select is slower?

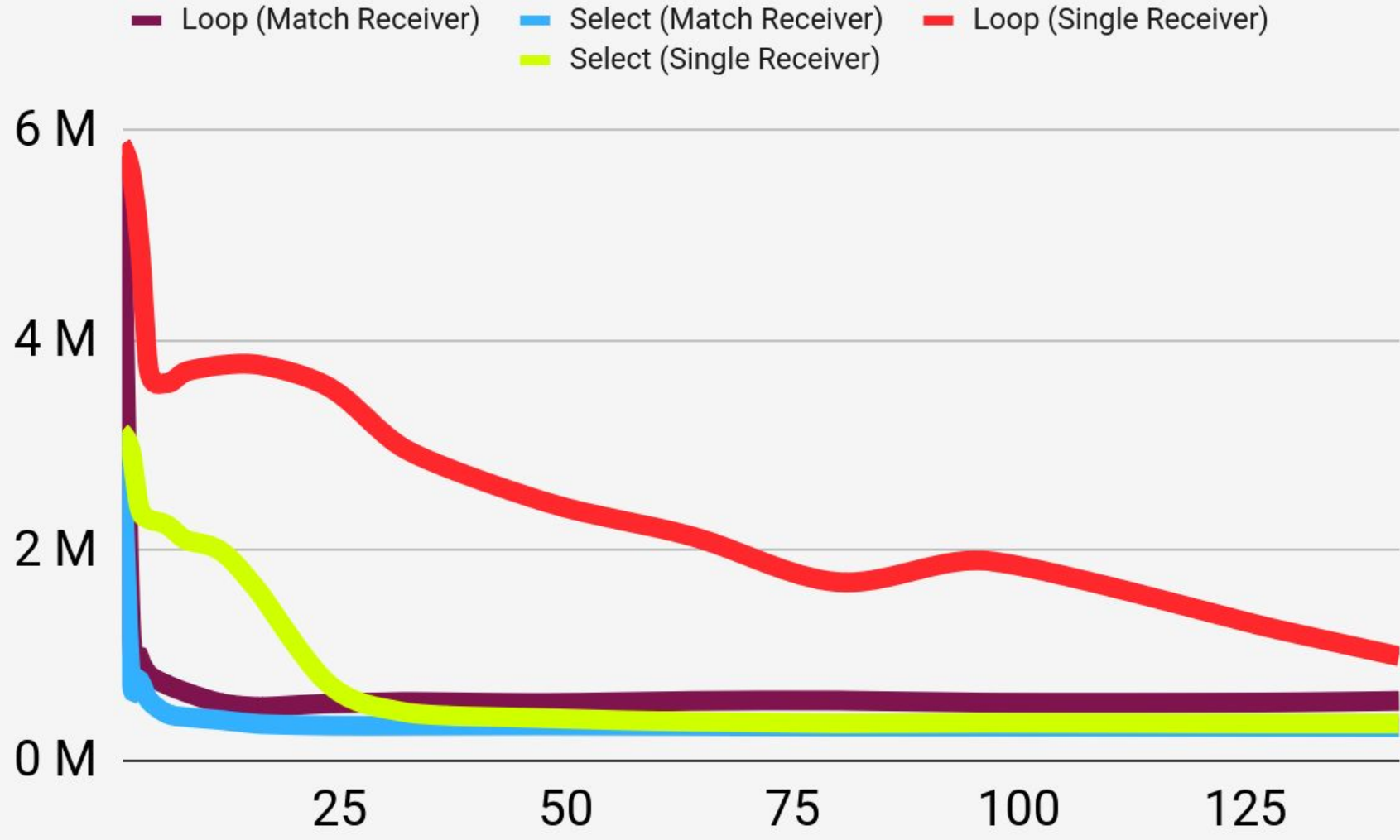


Select has more contention

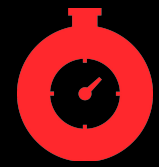


Matched Receivers

Effect of Matching Senders & Receivers



Having a single worker can be faster

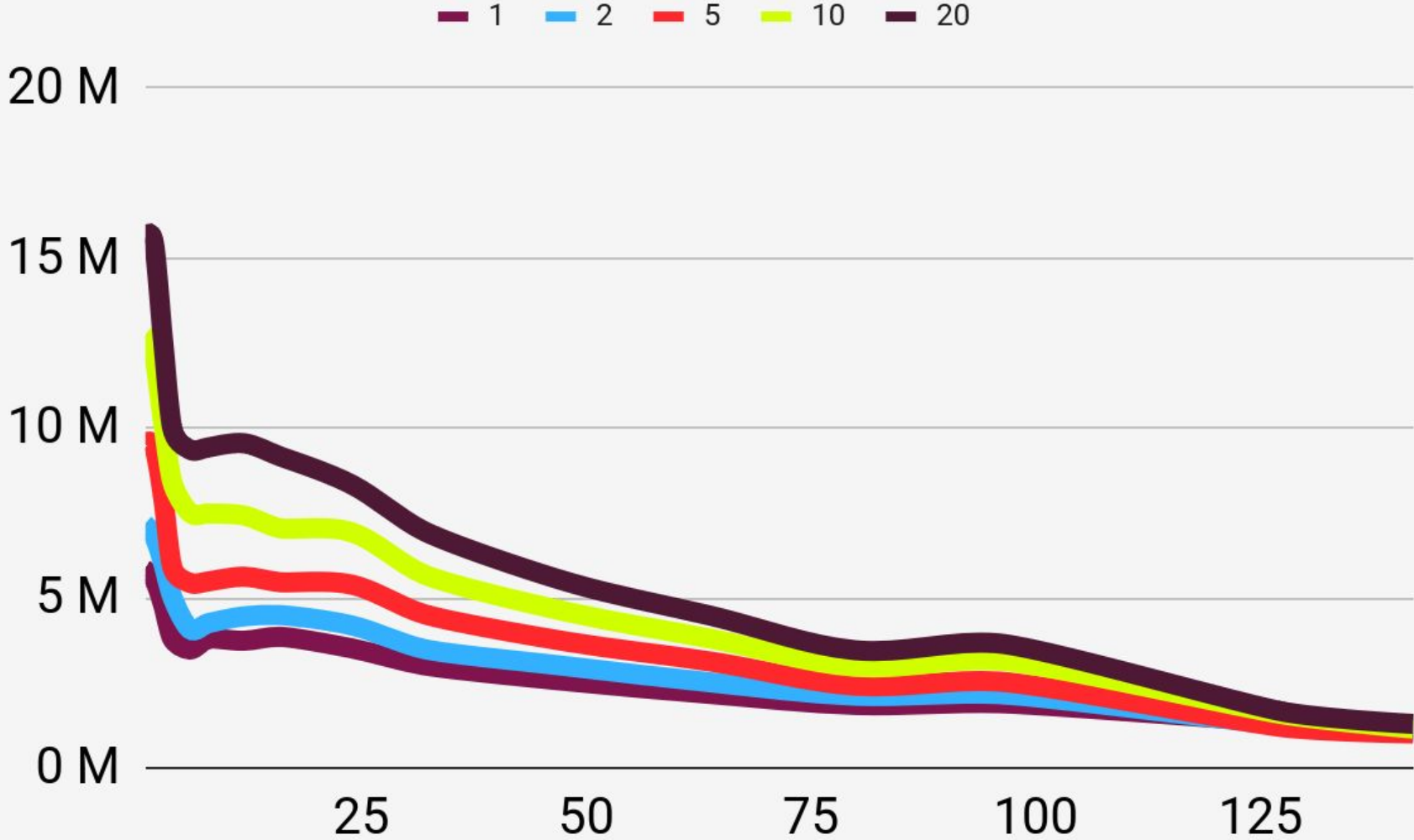


Having a single worker halves
the contention issue

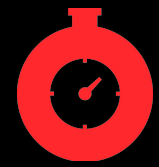


Buffers

Effect of Buffer size



Buffers help a bit at lower CPU counts



Buffers buy some headroom, but
it's basically all downhill

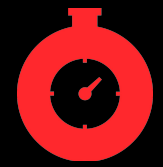


TL;DR

With a CPU count of more than 60, channel throughput is limited to around 1 million per second



Solutions



Solutions?

func GOMAXPROCS

```
func GOMAXPROCS(n int) int
```

GOMAXPROCS sets the maximum number of CPUs that can be executing simultaneously and returns the previous setting. It defaults to the value of `runtime.NumCPU`. If $n < 1$, it does not change the current setting. This call will go away when the scheduler improves.

Happy medium seems to be around 32
Very workload dependent

func LockOSThread

```
func LockOSThread()
```

LockOSThread wires the calling goroutine to its current operating system thread. The calling goroutine will always execute in that thread, and no other goroutine will execute in it, until the calling goroutine has made as many calls to [UnlockOSThread](#) as to LockOSThread. If the calling goroutine exits without unlocking the thread, the thread will be terminated.

All init functions are run on the startup thread. Calling LockOSThread from an init function will cause the main function to be invoked on that thread.

A goroutine should call LockOSThread before calling OS services or non-Go library functions that depend on per-thread state.

Unfortunately don't have control over which thread

Timeout

At some point it could be holding everything up, so have a way to exit early.

```
select {  
  case ch <- ent:  
  case <-timeout.C:  
    // do something with this  
}
```

Scale the channels

Hard coded, or auto scale them.

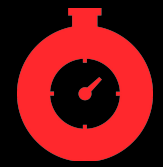
Adds a lot of complexity

```
chIdx := time.Now().Nanosecond() % maxBufCount
select {
case chs[bufIdx].Load().(chan msgType) <- ent:
    ...
```

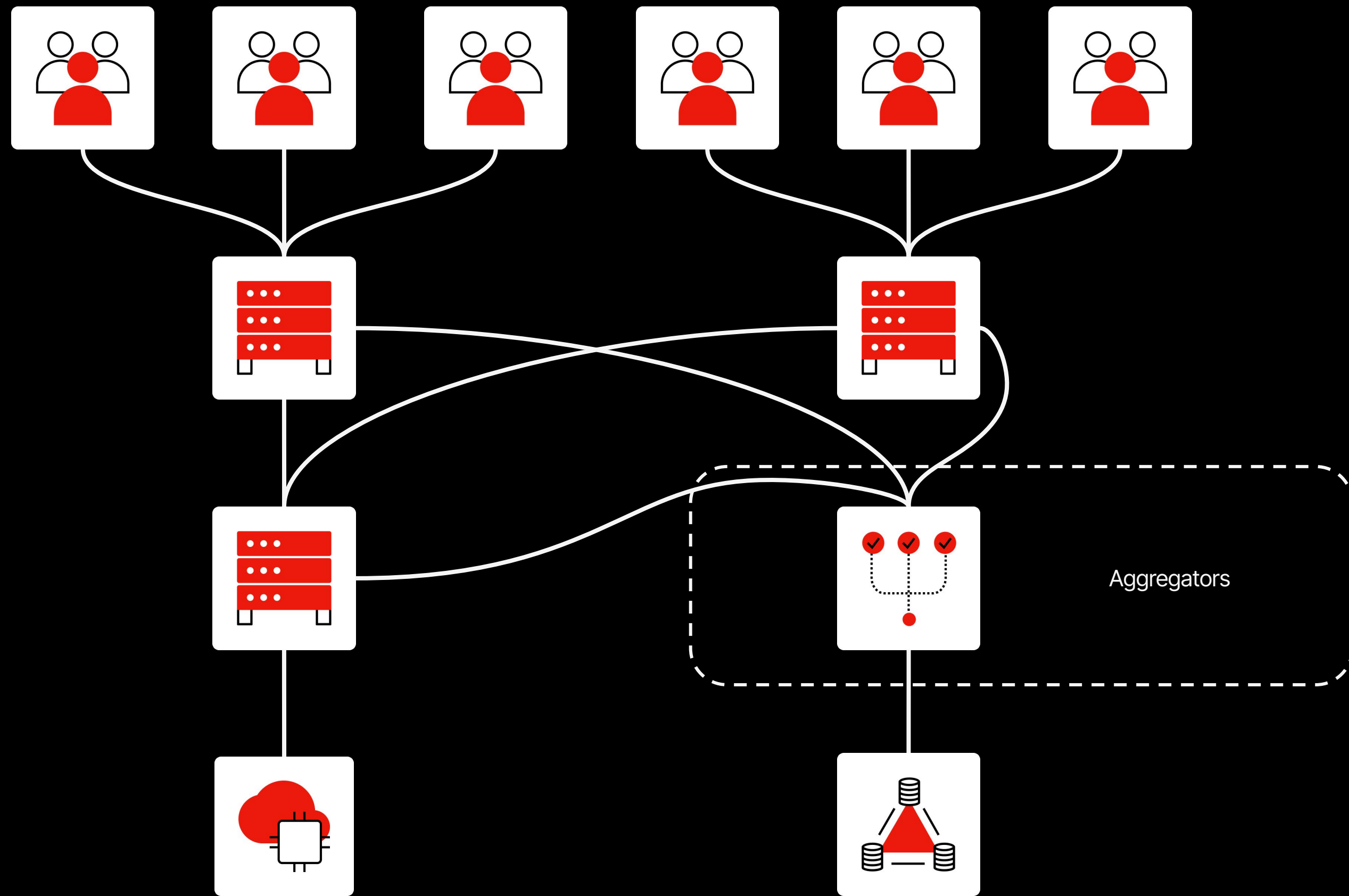
Buffers

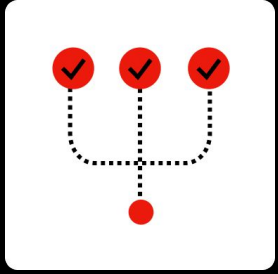
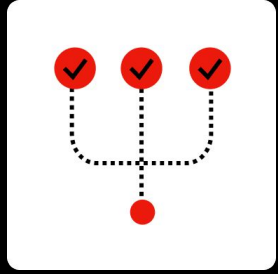
Reduces writes to the channel by the size of the buffer, but does introduce locking.

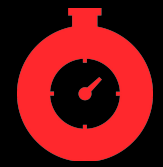
```
b.Lock()
buffer = append(buffer, ent)
if len(buffer) == maxBufSize {
    ch <- buffer
    buffer = buffer[:0]
}
b.Unlock()
```



Containers!?







Conclusions



TL;DR

With a CPU count of more than 60, channel throughput is limited to around 1 million per second



TL;DR

Solutions include:
GOMAXPROCS
Buffered Channels
Scaling Channels
Timeouts

Links

Used in this presentation:

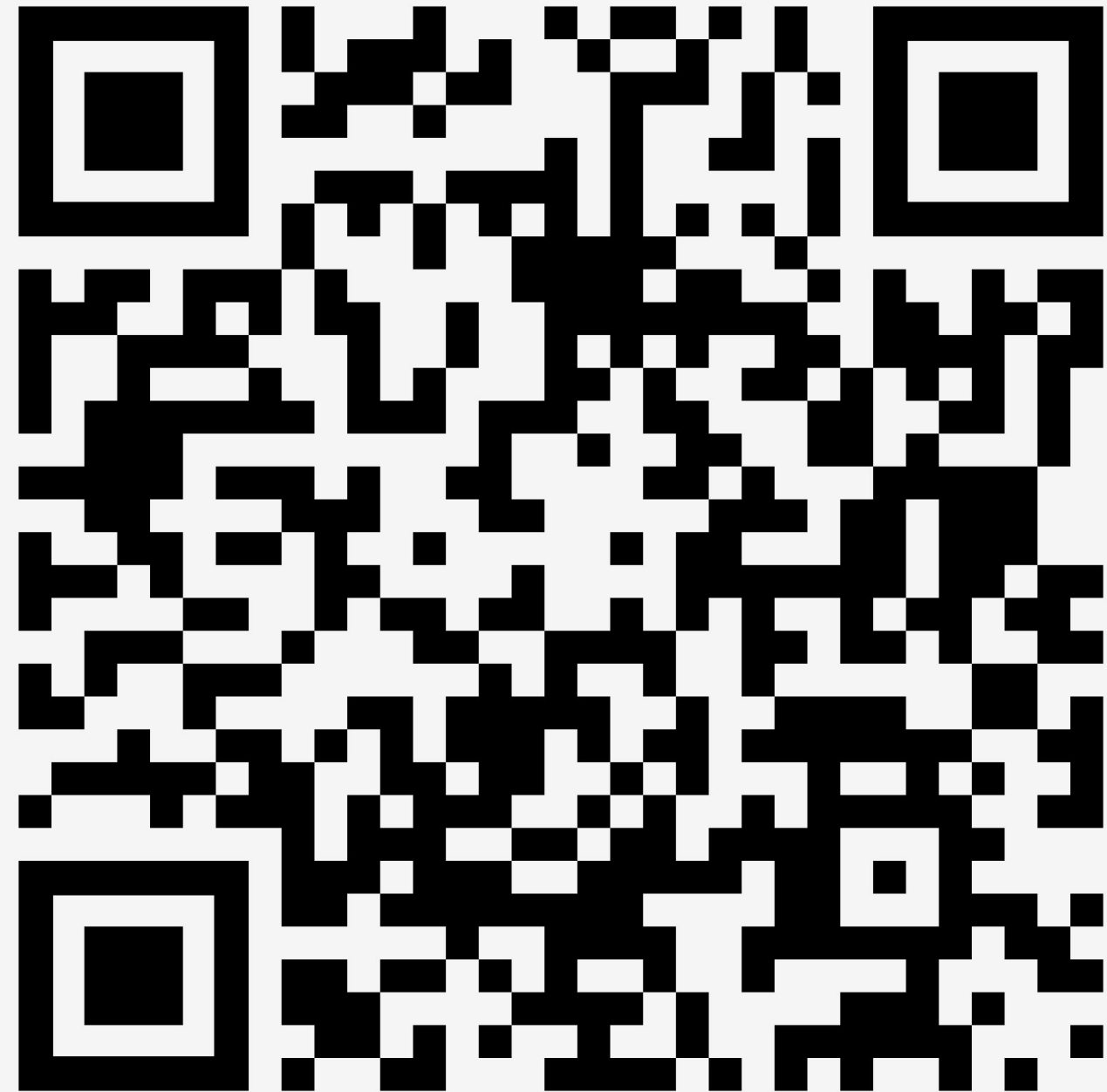
- [Runtime Metrics](#)
- [So just how fast are channels anyway?](#)
- [How to write benchmarks in Go](#)
- [Benchstat](#)
- [GOMAXPROCS](#)
- [LockOSThread](#)

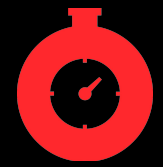
Me:

- [@rexfuzzle@hub13.xyz](#)
- [https://www.linkedin.com/in/grantstephensza](#)
- [grant@stephens.co.za](#)

This presentation:

- [https://exactly-right-airedale.edgecompute.app/](#)





Thank you