A study of Go and its ecosystem

Agenda

- What does it mean to be robust?
- Robust features of Go
- Fragile features of Go
- Giving up
- Well, actually: Erlang
- A new hope

slides

goto; copenhagen

GOTO Copenhagen 2018

Conference Nov. 19 - 21

Click 'Rate Session' to rate session and ask questions.

Follow us @gotocph

Francesc Campoy

VP of Product & DevRel

francesc@sourced.tech

@francesc

github.com/campoy

source{d}

source{d}

Machine Learning for Large Scale Code Analysis

Code as Data

Turn your code into actionable insights

LEARN MORE

Machine Learning on Code

Empower your Developers with Assisted Code Review

LEARN MORE

later today

Wednesday Nov 21 14:30 – 15:20
Location: Aud 10+11

Machine Learning on Source Code

source{d} is building the open-source components to enable large-scale code analysis and machine learning on source code.

Their powerful tools can ingest all of the world's public git repositories turning code into ASTs ready for machine learning and other analyses, all exposed through a flexible and friendly API.

Francesc Campoy, VP of Developer Relations at source{d}, will show you how to run machine learning on source code with a series of live demos.

machine learning | ai | live demo

Francesc Campoy
Gopher, Host of the @justforfunc podcast, and VP of Developer relations at source{d}

Website | Blog

What does it mean to be robust?


Robustness


Error Error
An error occurred while displaying the previous error.
OK

Fragility

Robust features of Go

Memory safety

- Pointers for convenience, but no pointers arithmetics.
- Escape analysis for automated allocation on heap/stack.
- Garbage collection: no dangling pointers.
- Automatic bound checks for slices and arrays.
  - Negative indices are forbidden, avoiding a whole class of errors.
  - No buffer overflow (unless bug in the language ...)

This makes memory corruption **basically** impossible.

Stack allocation (important for performance)

```
func value() int {
    v := new(int)   // allocated on the stack,
                    // because v doesn't escape.
    return *v
}

func main() {  fmt.Println(value()) }
```

Heap allocation (important for correctness)

```
func value() *int {
    v := 42          // allocated on the heap,
                     // because v escapes
    return &v
}

func main() {  fmt.Println(*value()) }
```

Bound checks (important for correctness)

```
func main() {
    a := make([]int, 256)
    a[512] = 42 // panic: runtime error: index out of range
}
```

Type safety

- Static typing
- Explicit type conversion for numeric types

```
int64 + int32 // mismatched types int32 and int64
```

- No unsafe implicit conversions, no automatic type coercion

```
42 + “hello”   // mismatched types int and string
               // not “42hello”
```

Type safety

- Compile-time but implicit interface satisfaction

```
v := 42
fmt.Fprintln(v, “hello”)
    // cannot use v (type int) as type io.Writer in argument to fmt.Fprintln:
    //   int does not implement io.Writer (missing Write method)
```

- Interfaces keep the type of the stored value

```
var i interface{} = v
i.(string) // panic: interface conversion: interface {} is int, not string
```

Unused variables; the compilation error

Seems surprising, but it has caught more than one bug!

```
var s [][]int
for i, row := range s {
    for j, cell := range row { // cell declared and not used
        cell = i * j
    }
}
```

Errors are not exceptional

- Go doesn’t have exceptions
- Exceptions are banned in C++ code at Google
- The main reason is that exceptions break the linear flow of a program, causing subtle bugs.

GopherCon 2016: Dave Cheney - Dont Just Check Errors Handle Them Gracefully

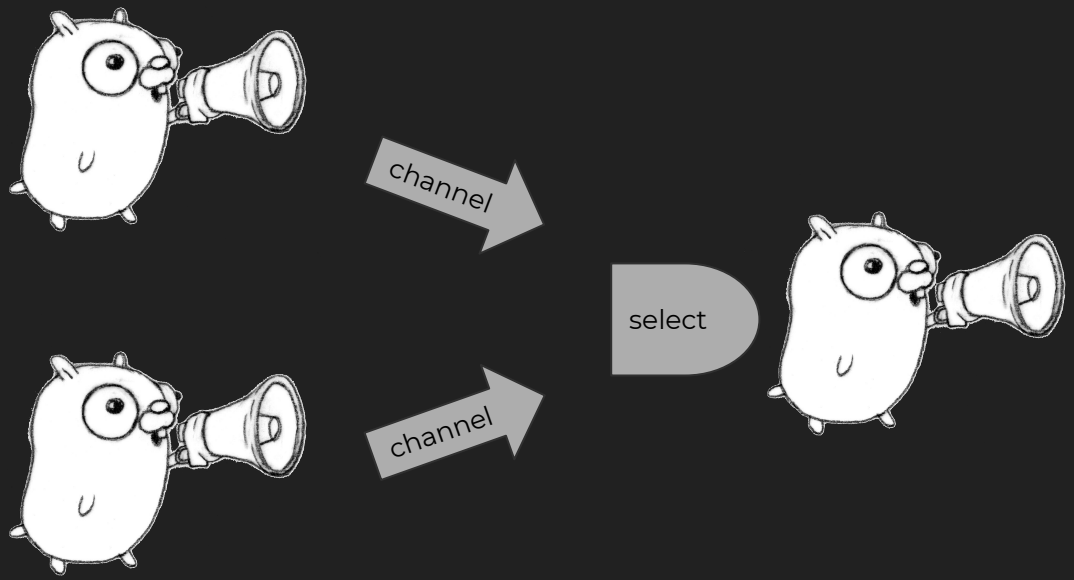
A subtle bug

```
var mutex sync.Mutex

func withMutex(f func()) {
    mutex.Lock()
    f()
    mutex.Unlock()
}
```

Channels

A simpler concurrency model makes it easier to implement correct patterns.



Fragile features of Go

Mutable shared state

```
var counter int

func ticker() {
  for range time.Tick(time.Second) {
      log.Printf("counter is %d\n", counter)
  }
}

func count(w http.ResponseWriter, r *http.Request) {
  counter++
}
```

Mutable shared state

```
func main() {
  go ticker()
  http.HandleFunc("/count", count)
  log.Fatal(http.ListenAndServe(":8080", nil))
}
```

```
$ go run main.go
2018/04/26 07:01:13 counter is 0
2018/04/26 07:01:14 counter is 1
…
```

Mutable shared state: tooling

Data race detector to the rescue!

```
$ go run -race main.go
2018/04/26 07:00:59 counter is 0
==================
WARNING: DATA RACE
Read at 0x000001581488 by goroutine 6:
  main.ticker()
Previous write at 0x000001581488 by goroutine 8:
  main.count()
```

Nil pointers!

Technically not that bad … but still a source of problems

Nil receivers, nil slices … but also nil maps

Lack of generics (yes I went there)

Monads are a great way to manage error

But without generics they’re quite hard to implement

Panic; then recover

Similar to exceptions, but used only “exceptionally”

```
func main() {
        defer func() {
                if err := recover(); err != nil {
                        // handle err
                }
        }()
    doStuff()
}
```

panic

```
func main() {
    http.HandleFunc("/", handler)
    log.Fatal(http.ListenAndServe(":8080", nil))
}

func handler(w http.ResponseWriter, r *http.Request) {
    panic("boo!")
}
```

$ go run server.go

```
2018/04/18 11:37:40 http: panic serving [::1]:56732: boo!

goroutine 5 [running ]:
net/http.(*conn).serve.func1(0xc420098820)
        /Users/francesc/go/src/net/http/server.go:1726 +0xd0
panic(0x12387a0, 0x12cdb70)
        /Users/francesc/go/src/runtime/panic.go:505 +0x229
main.handler(0x12d1800, 0xc420134000, 0xc42011e000)
        /Users/francesc/src/github.com/campoy/samples/recover/server.go:14 +0x39
net/http.HandlerFunc.ServeHTTP(0x12b0220, 0x12d1800, 0xc420134000, 0xc42011e000)
        /Users/francesc/go/src/net/http/server.go:1947 +0x44
net/http.(*ServeMux).ServeHTTP(0x140a3e0, 0x12d1800, 0xc420134000, 0xc42011e000)
        /Users/francesc/go/src/net/http/server.go:2337 +0x130
net/http.serverHandler.ServeHTTP(0xc42008b2b0, 0x12d1800, 0xc420134000, 0xc42011e000)
        /Users/francesc/go/src/net/http/server.go:2694 +0xbc
net/http.(*conn).serve(0xc420098820, 0x12d1a00, 0xc42010a040)
        /Users/francesc/go/src/net/http/server.go:1830 +0x651
created by net/http.(*Server).Serve
        /Users/francesc/go/src/net/http/server.go:2795 +0x27b
```

panic

```
func main() {
    http.HandleFunc("/", handler)
    log.Fatal(http.ListenAndServe(":8080", nil))
}

func handler(w http.ResponseWriter, r *http.Request) {
    go panic("boo!")
}
```

$ go run server.go

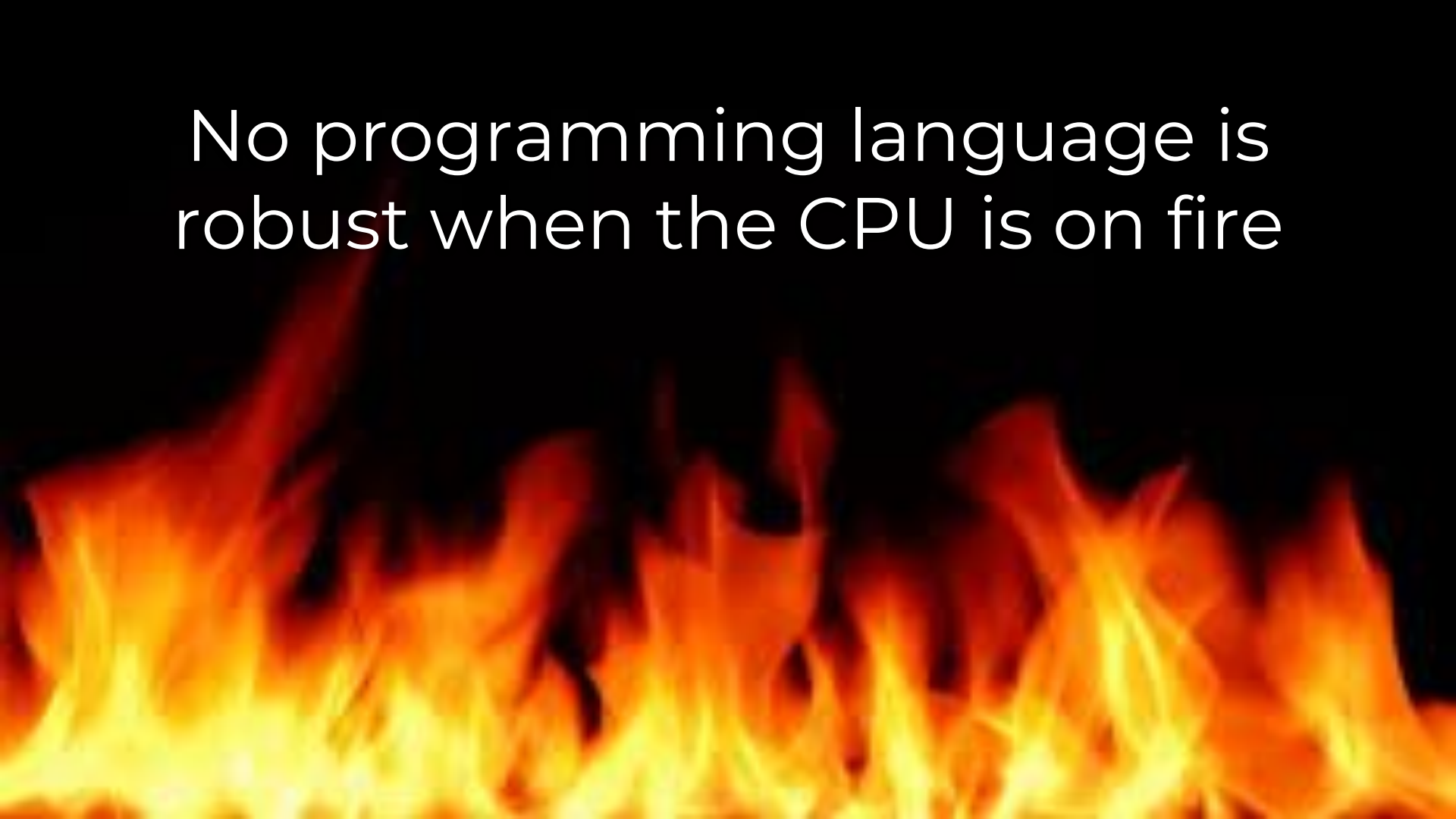
```
panic: boo!

goroutine 8 [running]:
panic(0x12387e0, 0xc420010b90)
        /Users/francesc/go/src/runtime/panic.go:554 +0x3c1
runtime.goexit()
        /Users/francesc/go/src/runtime/asm_amd64.s:2361 +0x1
created by main.handler
        /Users/francesc/src/github.com/campoy/samples/recover/server.go:14 +0x64

exit status 2
```


Giving up on Robustness

No programming language is robust when the CPU is on fire



Well, actually: Erlang

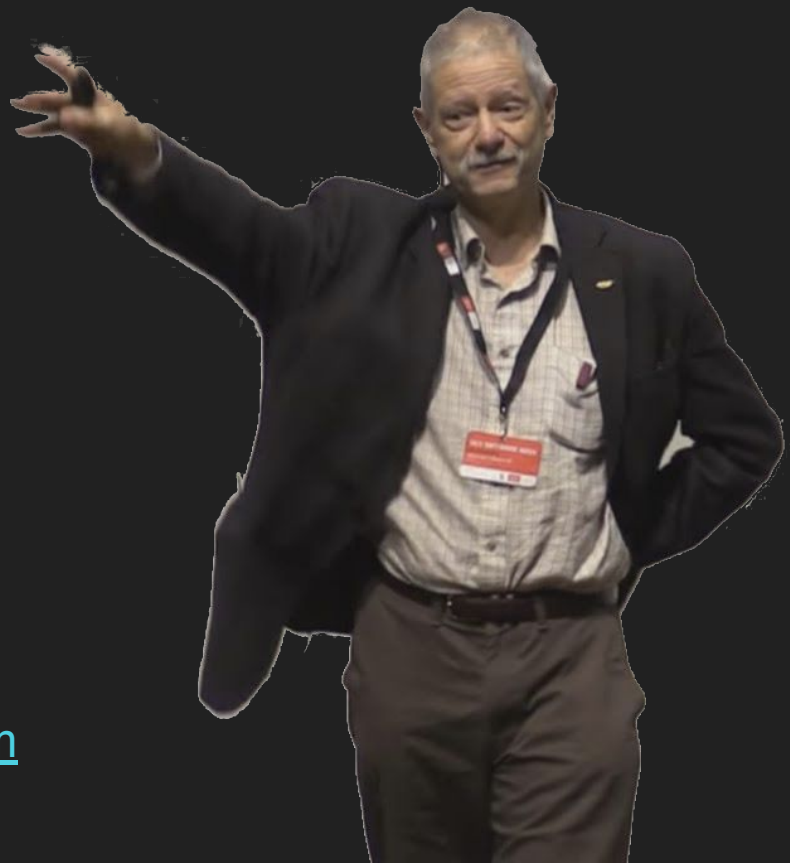
ERLANG

Systems that Run Forever Self-heal and Scale

Six rules:

1. Isolation
2. Concurrency
3. Failure detection
4. Fault Identification
5. Live Code Upgrade
6. Stable Storage

infoq.com/presentations/self-heal-scalable-system



I - Isolation

II - Concurrency

III - Failure Detection

IV - Fault Identification

V - Live Code Upgrade

VI - Stable Storage


“Let it crash”

Erlang vs. Go

	Erlang	Go
Isolation	✓	✗
Concurrency	✓	✓
Failure Detection	✓	✓
Fault Identification	✓	✗
Live Code Upgrade	✓	✗
Stable Storage	✓	✓

A new hope

kubernetes

I - Isolation

- Containers
- Namespaces
- Multiple nodes
- Multiple clusters / federation

II - Concurrency

- Go’s concurrency is great
- Extra Parallelism via replication
    - Replica controllers

III - Failure Detection

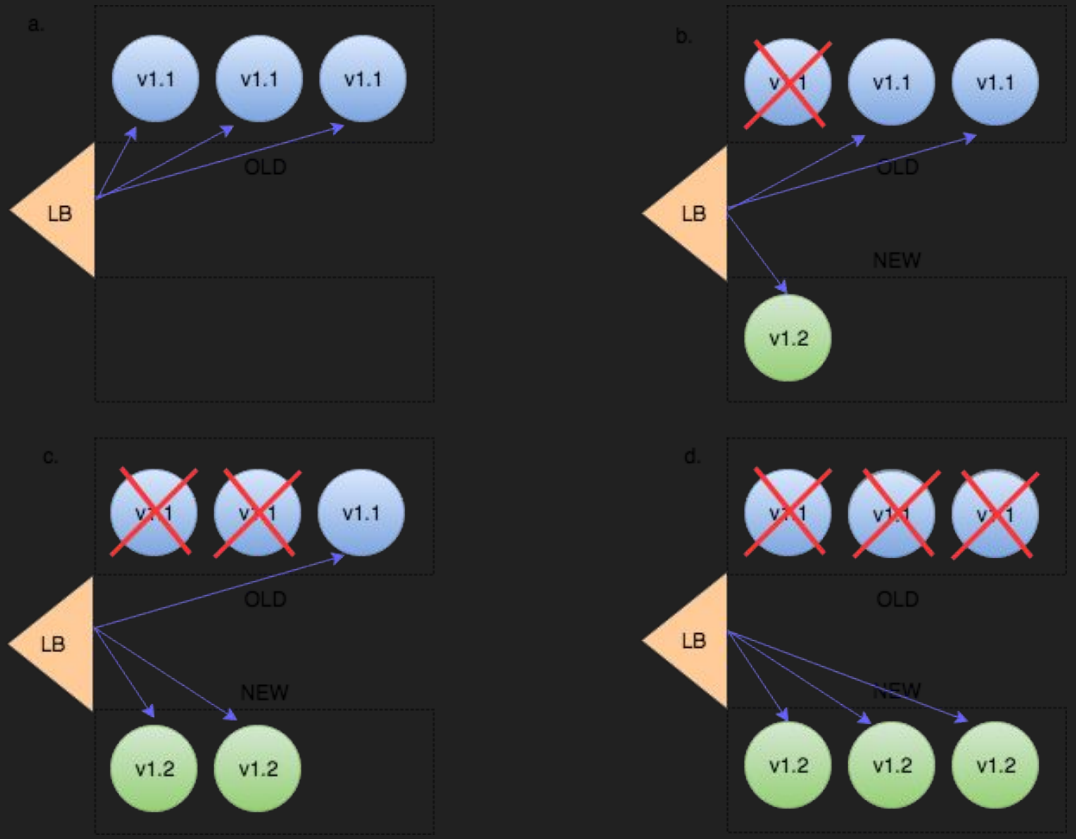
- Heartbeats (probes)
- Automated Monitoring
- Restart Policies

IV - Fault Identification

- Logs (but who reads them)
- /dev/termination-log

V - Live Code Upgrade

- Liveness Probes
- Readiness Probes
- Live Rolling Update



Kubernetes Rolling Update

VI - Stable Storage

- Not necessarily part of the system
  - Etcd
  - SQL databases
  - etc

Conclusion

- What does it mean to be robust?
- Robust features of Go
- Fragile features of Go
- Giving up
- Well, actually: Erlang
- A new hope

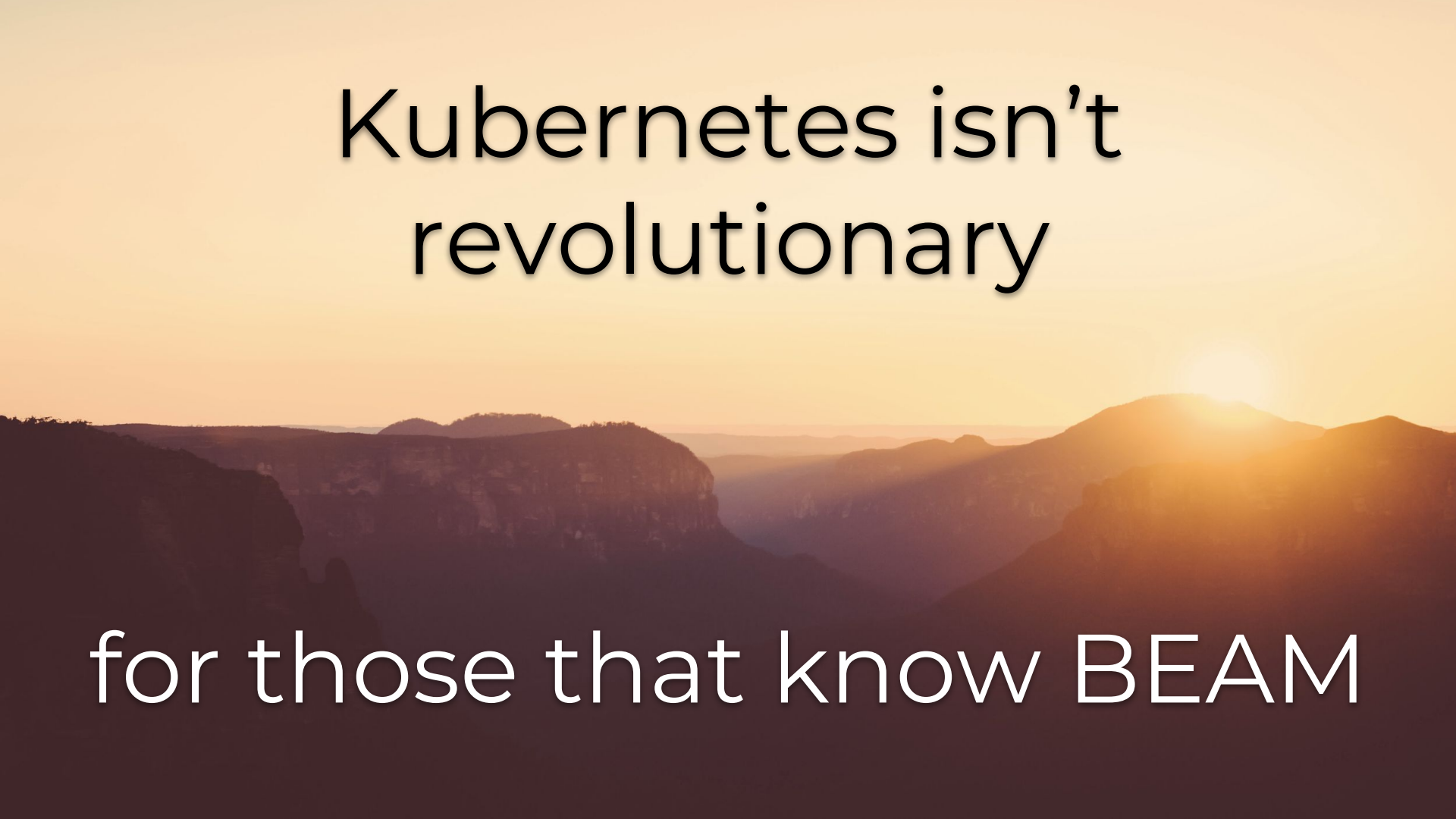
Erlang vs. Go

	Erlang	Go
Isolation	✓	✗
Concurrency	✓	✓
Failure Detection	✓	✓
Fault Identification	✓	✗
Live Code Upgrade	✓	✗
Stable Storage	✓	✓

Erlang vs. Go

	Erlang+BEAM	Go+K8s
Isolation	✓	✓
Concurrency	✓	✓
Failure Detection	✓	✓
Fault Identification	✓	✓
Live Code Upgrade	✓	✓
Stable Storage	✓	✓

Kubernetes isn’t revolutionary


Kubernetes isn’t
revolutionary
for those that know BEAM

Thanks!

@maria_fibonacci

@miriampena

BANNED

ERLANG

0:20 / 35:38

Miriam Pena - Keynote: Unsung Heroes of the BEAM - Code BEAM SF 2018

11:23 / 36:16

ElixirConf 2017 - My Journey from Go to Elixir - Veronica Lopez

goto; copenhagen

GOTO Copenhagen 2018
Conference Nov. 19 - 21

Please

Remember to rate this session

Thank you!

Follow us @gotocph

Thanks!

Francesc Campoy

@francesc
francesc@sourced.tech