

Successful Go for microservices architecture

Hervé Ah-Leung

Who?



Who am I?

Senior Software engineer in London

Smartnumbers since 2019

(<https://smartnumbers.com>)



Microservices?



Monolith vs Microservices

Monolith

- Gigantic mono-service
- Single database
- Unique repo

- > difficult to extend/maintain
- > often poor dev experience



Microservices

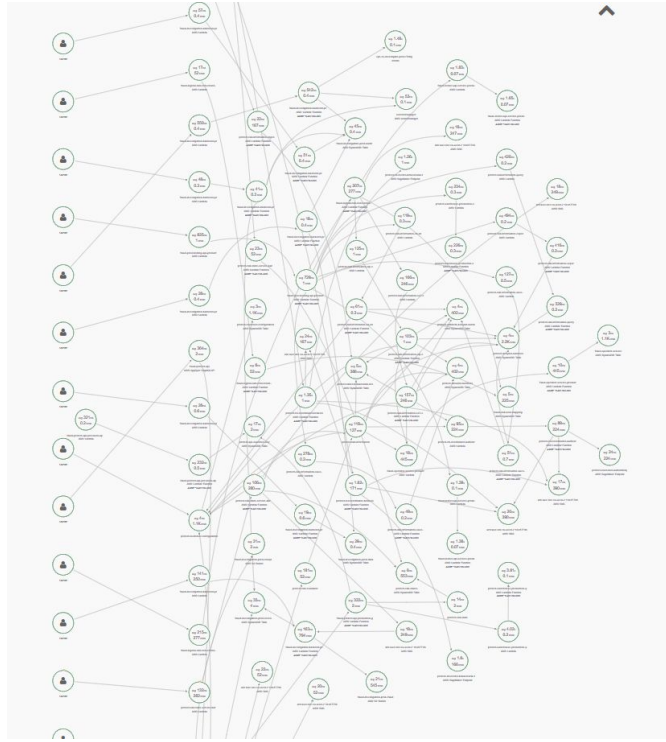
- Multiple autonomous meaningful services
- With their own databases
- In different repos

- > scalable and easy to extend
- > better dev experience



Microservices disadvantages/challenges

Service map AWS X-Ray (March 2022)



Microservices disadvantages/challenges

- Complexity of communication between services
- Increase of latency if not careful
- Easily extendable/maintainable? yes and no

> Any programming language could help solving/mitigating those problems, but how do we make the most of Go to tackle them?

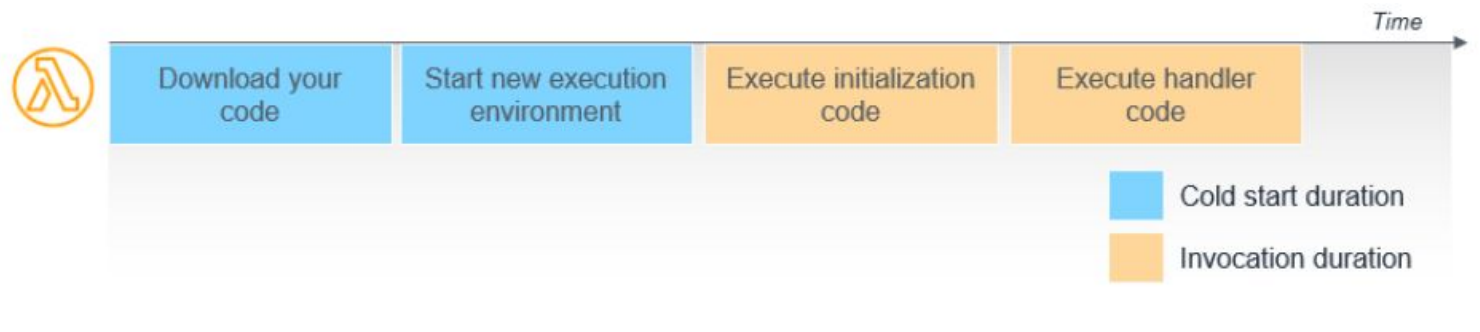


Let's take full advantage of Go!



Amazing serverless experience

Cold start problem 3 years ago with Scala with serverless functions



Amazing serverless experience

Cold start depends on:

- the language
- the package size
- If inside a private network or not

Thorough article:

<https://mikhail.io/serverless/coldstarts/aws/>



Amazing serverless experience

Observations:

- awful cold starts with Scala (Java): 1 to 2s
- much better with Go: ms

Learning

- Go binaries are lightweight, therefore the provisioning time is small
- First success for our microservices for latency



A well-supported and popular language

- Often Go SDKs available (AWS, GCP, Azure...)
- Plenty of tutorials available
- Living ecosystem and supportive community
- Living language (releases/improvements)

Learning

- Popularity of language massively helped to build smoothly the microservices



Excellent tooling

- Standard libraries!

▶ archive

bufio

builtin

bytes

▶ compress

▶ container

context

▶ crypto

▶ database

▶ debug

embed

▶ encoding

errors

expvar

flag

fmt

▶ go

▶ hash

▶ html

▶ image

▶ index

▶ io

▶ log

▶ math

▶ mime

▶ net

▶ os

▶ path

plugin

reflect

▶ regexp

▶ runtime

sort

strconv

strings

▶ sync

▶ syscall

▶ testing

▶ text

▶ time

▶ unicode

unsafe

▶ internal



- Dependencies management with inbuilt **Go modules** (we started with **dep**)



dep

Dependency management for Go

VS

Go Modules Reference

Table of Contents

Introduction	go mod graph
Modules, packages, and versions	go mod init
Module paths	go mod tidy
Versions	go mod vendor
Pseudo-versions	go mod verify
Major version suffixes	go mod why
Resolving a package to a module	go version -m
go.mod files	go clean -modcache
Lexical elements	Version queries
Module paths and versions	Module commands outside a module
Grammar	go work init
module directive	go work edit
go directive	go work use
require directive	go work sync
exclude directive	Module proxies
replace directive	GOPROXY protocol
retract directive	Communicating with proxies
Automatic updates	Serving modules directly from a proxy
Minimal version selection (MVS)	Version control systems
Replacement	Finding a repository for a module path
Exclusion	Mapping versions to commits
Upgrades	Mapping pseudo-versions to commits
Downgrade	Mapping branches and commits to versions
Module graph pruning	Module directories within a repository
Lazy module loading	Special case for LICENSE files
Workspaces	Controlling version control tools with GOVCS
go.work files	Module zip files
Lexical elements	File path and size constraints
Grammar	Private modules
go directive	Private proxy serving all modules
use directive	Private proxy serving private modules
replace directive	Direct access to private modules
Compatibility with non-module repositories	Passing credentials to private proxies
+incompatible versions	Passing credentials to private repositories
Minimal module compatibility	Privacy
Module-aware commands	Module cache
Build commands	Authenticating modules
Vendoring	go.sum files
go get	Checksum database
go install	Environment variables
go list -m	Glossary
go mod download	
go mod edit	

Excellent tooling

Testing

- **gomega**: matcher/assertion lib
- **ginkgo**: BDD test framework

Libraries

- **goReleaser**: binaries builder

GraphQL

- **gqlgen**: graphql server generator



Gomega

Ginkgo



 **gqlgen**

Excellent tooling

Quick special mention to Goland

A screenshot of the Goland IDE interface. The left sidebar shows a project tree with folders like 'basics', 'car', 'codewars', 'concurrency', 'context', 'design_pattern', 'builder', 'command', 'factory', 'observer', 'main.go', 'singleton', 'error', 'rails', 'testing', 'todo', 'controller.go', 'controller_test.go', 'database.json', 'identifier.go', 'main.go', 'persistence.go', 'todo.go', 'tricks', 'tube', 'weather', 'go.mod', and 'External Libraries'. The main editor window shows a Go file named 'scratch_3.go' with the following code:

```
1 package main
2
3 import "fmt"
4
5 func main() {
6     names := []string{"Gemma", "Sue", "Steven"}
7
8     for _, name := range names {
9         fmt.Printf(format:"Hello %s\n", name)
10     }
11 }
12
13
```



Learning

- Key libraries/tooling make the dev experience solid
 - Faster to build microservices
 - Reliable quality
 - Writing Go is enjoyable



New joiner experience

- Relatively easy to learn
 - A few keywords
 - Explicit friendly syntax
- Room to improve if challenges are needed
 - Garbage collection
 - Concurrency model
- A junior member even did a presentation to the whole engineering department about Go

A tour of Go
go.dev/tour

Go by example
gobyexample.com/



Learning

- Nice dev experience for a junior software engineer
- Rewarding learning curve
- Nice experience as a mentor
- Can focus on the microservices instead of language details



Any limitations?



Folders and packages organisation

Idea of contexts (not Go context...)

```
- controller
  - customer_controller.go
  - order_controller.go
  - sales_controller.go
- services
  - payment.go
  - shipment.go
  - authentication.go
  - refund.go
  - ...
- views
  - ...
- models
  - ...
```

VS

```
- order
  - service.go
  - controller.go
  - request.go
- customer
  - authentication.go
  - model.go
- refund
  - service.go
  - form.go
```



Learning

Organising in context helps to identify common packages and extract them as common libraries.

Tooling then do the rest.



Specificities of the language

Not necessarily impacting a microservices architecture, but worth mentioning

- Pointers
 - Null pointer exception leading to actual panic
- Interfaces
 - Implicit interfaces concept sometimes difficult to understand for a beginner



Next steps?



Wider Go usage?

As a Go software engineer

I want to write my **infrastructure** in Go

I want to write my **deployment pipeline** in Go

I want to write my **tasks runner** in Go

I want to write Go to generate my **documentation**

So then I don't write awful YAML



Protobuf?

- Structure data serialisation



Protocol Buffers

Home Guides Reference Support

Filter

Overview

Developer Guide

- Language Guide (proto2)
- Language Guide (proto3)
- Style Guide
- Encoding
- Techniques
- Adapters

Tutorials

- Tutorials Overview
- Basics: C++
- Basics: C#
- Basics: Dart
- Basics: Go**
- Basics: Java
- Basics: Kotlin
- Basics: Python

Related Guides

- gRPC



Go 1.18 is released!

The Go Team

15 March 2022

Today the Go team is thrilled to release Go 1.18, which you can get by visiting the [download page](#).

Go 1.18 is a massive release that includes new features, performance improvements, and our biggest change ever to the language. It isn't a stretch to say that the design for parts of Go 1.18 started over a decade ago when we first released Go.

Generics

In Go 1.18, we're introducing new support for [generic code using parameterized types](#). Supporting generics has been Go's most often requested feature, and we're proud to deliver the generic support that the majority of users need today. Subsequent releases will provide additional support for some of the more complicated generic use cases. We encourage you to get to know this new feature using our [generics tutorial](#), and to explore the best ways to use generics to optimize and simplify your code today. The [release notes](#) have more details about using generics in Go 1.18.

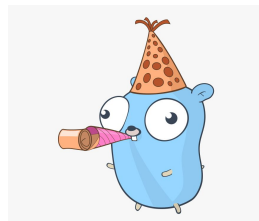


In conclusion?



Wrap up

- Main issues of a microservices architecture
 - Complexity of communication between services
 - Latency between services
- Go helps to ease the pain points
 - Excellent performances
 - Excellent tooling
 - Good popularity
 - Excellent dev experience
- Probably more Go features to come to build amazing software!



Thank you!

