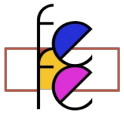
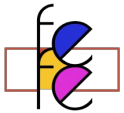


Continuous Profiling Go Application running in Kubernetes

@gianarb / gianarb.it



/debug/pprof/

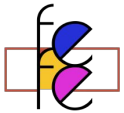
Types of profiles available:

Count	Profile
6797	allocs
358	block
0	cmdline
322	goroutine
6797	heap
164	mutex
0	profile
89	threadcreate
0	trace

full goroutine stack dump

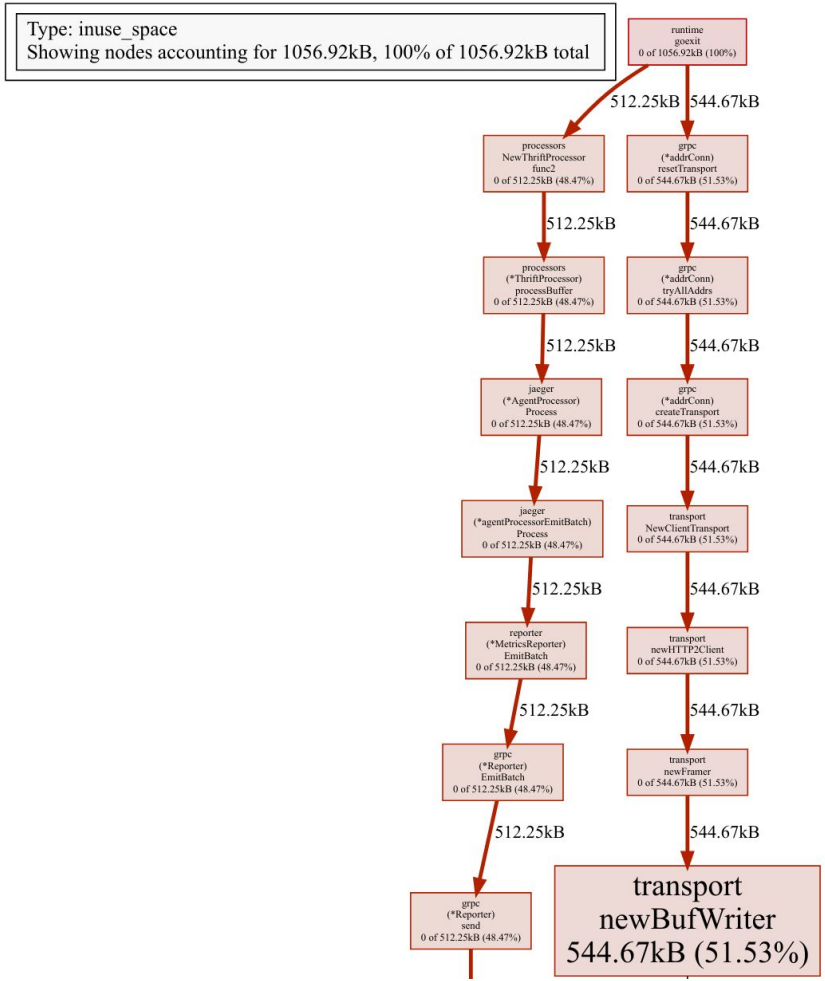
Profile Descriptions:

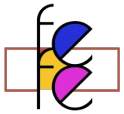
- allocs: A sampling of all past memory allocations
- block: Stack traces that led to blocking on synchronization primitives
- cmdline: The command line invocation of the current program
- goroutine: Stack traces of all current goroutines
- heap: A sampling of memory allocations of live objects. You can specify the gc GET parameter to run GC before taking the heap sample.
- mutex: Stack traces of holders of contended mutexes
- profile: CPU profile. You can specify the duration in the seconds GET parameter. After you get the profile file, use the go tool pprof command to investigate the profile.
- threadcreate: Stack traces that led to the creation of new OS threads
- trace: A trace of execution of the current program. You can specify the duration in the seconds GET parameter. After you get the trace file, use the go tool trace command to investigate the trace.



$ go tool pprof http://localhost:14271/debug/pprof/allocs?debug=1

```
Fetching profile over HTTP from http://localhost:14271/debug/pprof/allocs?debug=1
Saved profile in /home/gianarb/pprof/pprof.alloc_objects.alloc_space.inuse_objects.inuse_space.001.pb.gz
Type: inuse_space
Entering interactive mode (type "help" for commands, "o" for options)
(pprof) text
Showing nodes accounting for 1056.92kB, 100% of 1056.92kB total
Showing top 10 nodes out of 21
      flat  flat%   sum%        cum   cum%
  544.67kB 51.53% 51.53%   544.67kB 51.53%  github.com/jaegertracing/jaeger/vendor/google.golang.org/grpc/internal/transport.newBufWriter
  512.25kB 48.47%   100%   512.25kB 48.47%  time.startTimer
         0     0%   100%   512.25kB 48.47%  github.com/jaegertracing/jaeger/cmd/agent/app/processors.(*ThriftProcessor).processBuffer
         0     0%   100%   512.25kB 48.47%  github.com/jaegertracing/jaeger/cmd/agent/app/processors.NewThriftProcessor.func2
         0     0%   100%   512.25kB 48.47%  github.com/jaegertracing/jaeger/cmd/agent/app/reporter.(*MetricsReporter).EmitBatch
         0     0%   100%   512.25kB 48.47%  github.com/jaegertracing/jaeger/cmd/agent/app/reporter/grpc.(*Reporter).EmitBatch
         0     0%   100%   512.25kB 48.47%  github.com/jaegertracing/jaeger/cmd/agent/app/reporter/grpc.(*Reporter).send
         0     0%   100%   512.25kB 48.47%  github.com/jaegertracing/jaeger/proto-gen/api_v2.(*collectorServiceClient).PostSpans
         0     0%   100%   512.25kB 48.47%  github.com/jaegertracing/jaeger/thrift-gen/jaeger.(*AgentProcessor).Process
         0     0%   100%   512.25kB 48.47%  github.com/jaegertracing/jaeger/thrift-gen/jaeger.(*agentProcessorEmitBatch).Process
```


Type: inuse_space
Showing nodes accounting for 1056.92kB, 100% of 1056.92kB total
runtime
goexit
0 of 1056.92kB (100%)
512.25kB
544.67kB
processors
NewThriftProcessor
func2
0 of 512.25kB (48.47%)
grpc
(*addrConn)
resetTransport
0 of 544.67kB (51.53%)
512.25kB
544.67kB
processors
(*ThriftProcessor)
processBuffer
0 of 512.25kB (48.47%)
grpc
(*addrConn)
tryAllAddrs
0 of 544.67kB (51.53%)
512.25kB
544.67kB
jaeger
(*AgentProcessor)
Process
0 of 512.25kB (48.47%)
grpc
(*addrConn)
createTransport
0 of 544.67kB (51.53%)
512.25kB
544.67kB
jaeger
(*agentProcessorEmitBatch)
Process
0 of 512.25kB (48.47%)
transport
NewClientTransport
0 of 544.67kB (51.53%)
512.25kB
544.67kB
reporter
(*MetricsReporter)
EmitBatch
0 of 512.25kB (48.47%)
transport
newHTTP2Client
0 of 544.67kB (51.53%)
512.25kB
544.67kB
grpc
(*Reporter)
EmitBatch
0 of 512.25kB (48.47%)
transport
newFramer
0 of 544.67kB (51.53%)
512.25kB
544.67kB
grpc
(*Reporter)
send
0 of 512.25kB (48.47%)
transport
newBufWriter
544.67kB (51.53%)



Overview ▾

Package pprof serves via its HTTP server runtime profiling data in the format expected by the pprof visualization tool.

The package is typically only imported for the side effect of registering its HTTP handlers. The handled paths all begin with /debug/pprof/.

To use pprof, link this package into your program:

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

If your application is not already running an http server, you need to start one. Add "net/http" and "log" to your imports and the following code to your main function:

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

Then use the pprof tool to look at the heap profile:

```
go tool pprof http://localhost:6060/debug/pprof/heap
```

Gianluca Arbezzano

Software Engineer sold to reliability @InfluxData

- https://gianarb.it
- @gianarb

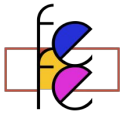
What I like:

- I make dirty hacks that look awesome
- I grow my vegetables 🍅🌻🍆
- Travel for fun and work

Applications make troubles in production

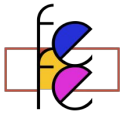
How developers extract profiles from production?

The common way is by bothering who better knows IPs
and how to connect to prod



Usually they have better thing to do than babysitting SWE

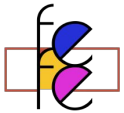
But it is not a SWE fault because they do not have a good way to retrieve what they need to be effective at their work.



you never know when you will need a profile, and for what or from where

Let’s summarize issues

- Developer are usually the profile stakeholder
- Production is not always a comfortable place to interact with
- You do not know when you will need a profile, it will may be from 2 weeks ago
- Cloud, Kubernetes increases the amount of noise. A lot more binaries, they go up and down continuously. Containers that OOMs gets restarted transparency, there is a lot of postmortem analysis going on



Do you have the same problem with your metrics/logs?!

Are you ready to know a possible solution?

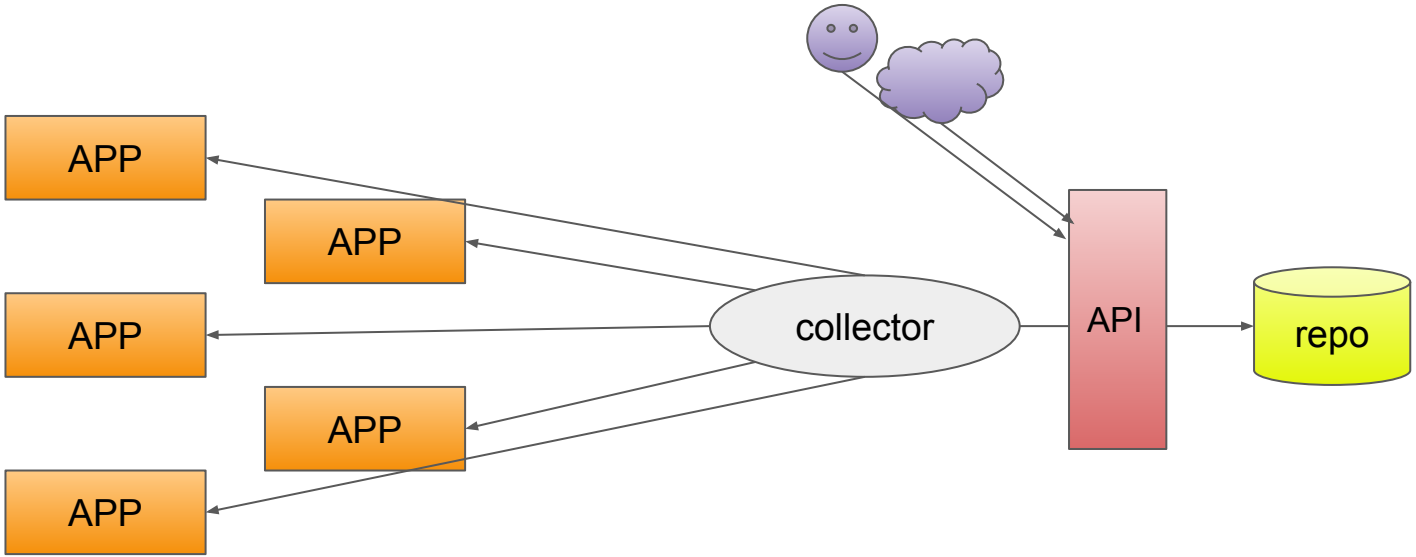
Spoiler Alert: it is part of the title

@gianarb / gianarb.it

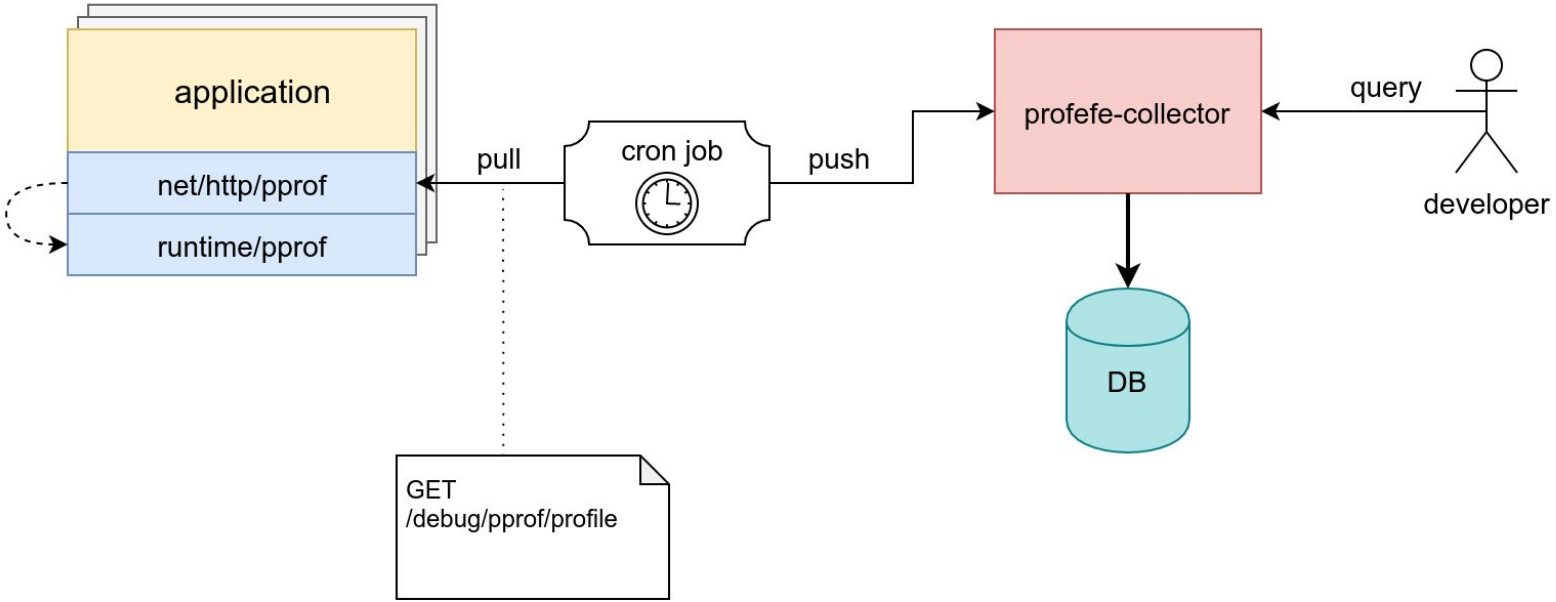
Metrics/Logs

They are **continuously** collected and stored in a **centralized** place.

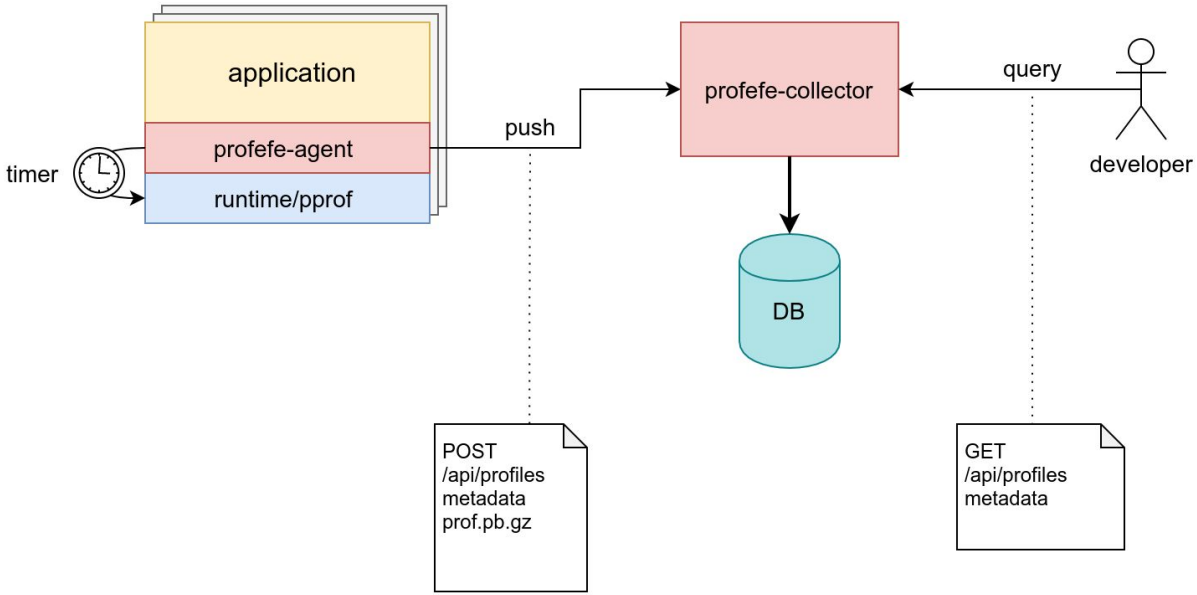
Follow me



github.com/profefe



github.com/profefe



The pull based solution was easier to implement for us:

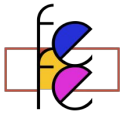
- Too many applications to re-instrument with the sdk
- Our services already expose pprof http handler by default

APP
APP
APP
APP
APP
APP
APP
APP
APP
APP
APP
APP
APP
APP

Kubernetes provides APIs!

@gianarb / gianarb.it

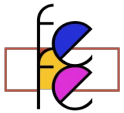
1 + 1 = 2



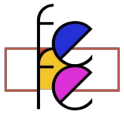
Let’s make a cronjob that uses the k8s api

github.com/profefe/kube-profefe

@gianarb / gianarb.it



Now profiles are continuously gathered from all your application

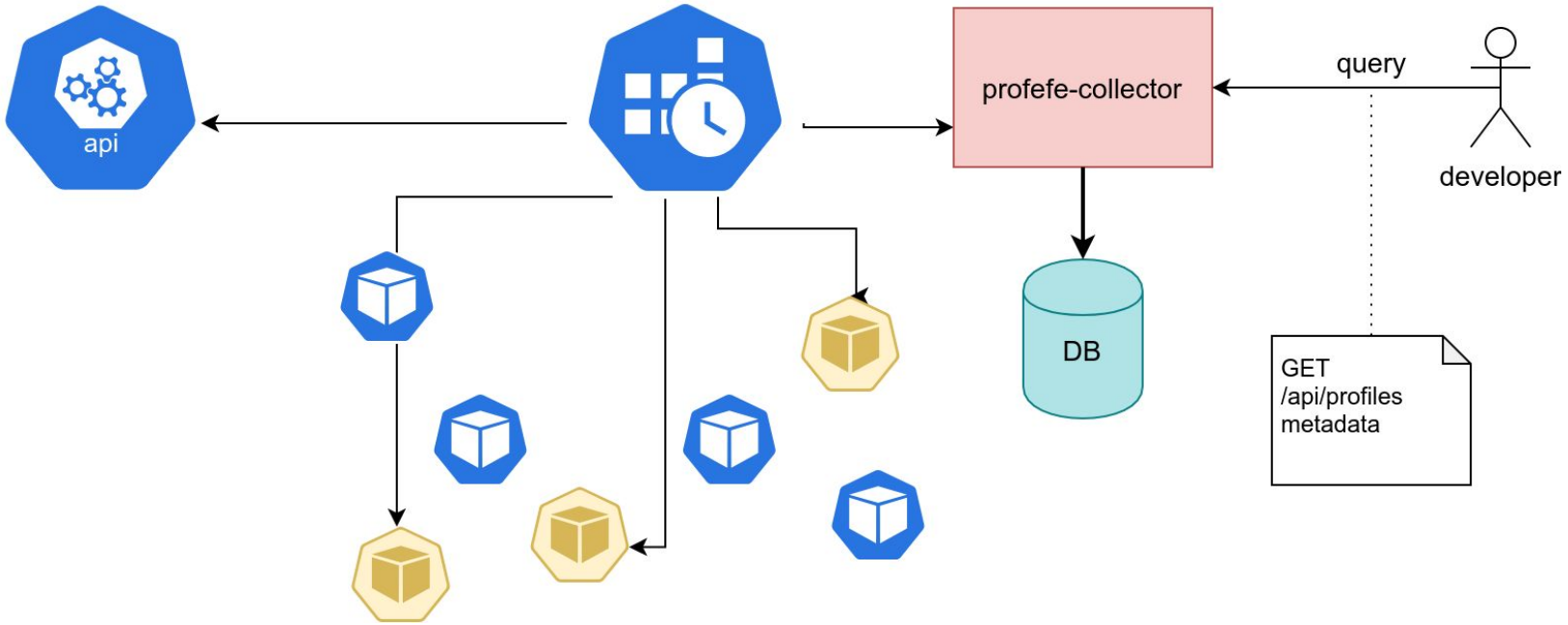


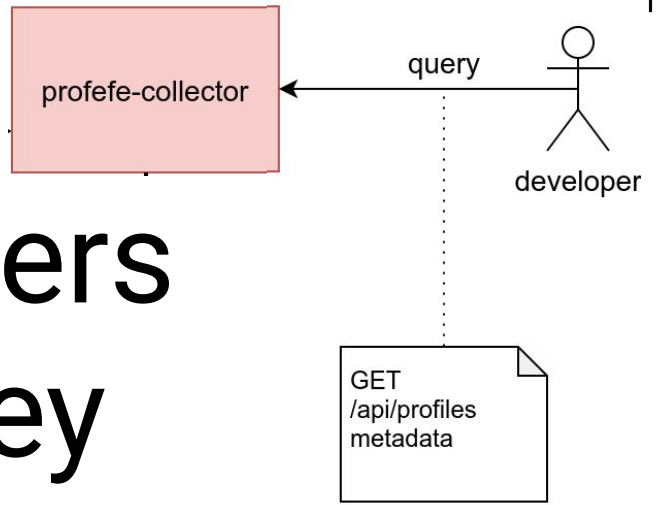
How it works

Golang has an http handler that exposes pprof over http, via annotation we can specify if a pod has profiles to capture and with other annotations we can configure path and port.

The annotations are:

- `profefe.com/enable=true` is the annotation that tells kube-profefe to capture profiles from that pod.
- `profefe.com/port=8085` tells kube-profefe where to look for a pprof http server. By default it is 6060.
- `profefe.com/service=frontend` tells kube-profefe the name of the service usable to lookup the profile. If the annotation is not specified it uses the pod name. My suggestion is to set this annotation because pods are ephemeral and lookup by pod name may be hard to do.
- `profefe.com/path=/debug/pprof` tells kube-profefe where to look for a pprof http server. By default it is `/debug/pprof`.


api
profefe-collector
query
developer
DB
GET
/api/profiles
metadata



How to let developers free to get what they want by themself?

$ kubectl profefe

$ kubectl profefe capture -n ops influxdb-v2

Cool things: Merge profile

go tool pprof
'http://repo.pprof.cluster.local:10100/api/0/profiles/merge?service=auth&type=cpu&**from=2019-05-30T11:49:00&to=2019-05-30T12:49:00**&labels=version=1.0.0'

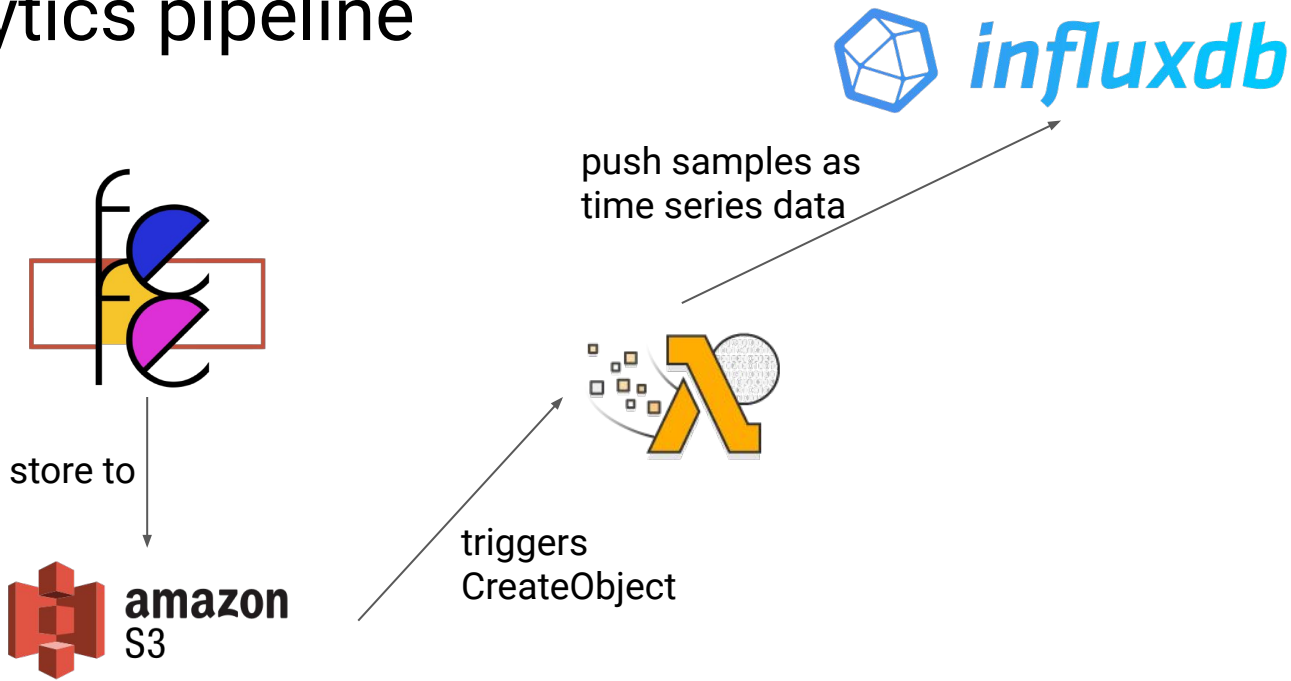
```
Fetching profile over HTTP from http://localhost:10100/api/0/profiles...
Type: cpu
Entering interactive mode (type "help" for commands, "o" for options)
(pprof) top
Showing nodes accounting for 43080ms, 99.15% of 43450ms total
Dropped 53 nodes (cum <= 217.25ms)
Showing top 10 nodes out of 12
      flat  flat%   sum%        cum   cum%
   42220ms 97.17% 97.17%    42220ms 97.17%  main.load
     860ms  1.98% 99.15%      860ms  1.98%  runtime.nanotime
         0     0% 99.15%    21050ms 48.45%  main.bar
         0     0% 99.15%    21170ms 48.72%  main.baz
```

Pod 150 *
6 =
---
900 pprof/hour

Analyze pprof profiles

- Easy correlation with other metrics such as mem/cpu usage
- All those profiles contains useful information
- Cross service utilization for performance optimization
  - Give me the top 10 cpu intensive function in all system
- Building bridges between dev and ops

Analytics pipeline



Links:

- https://github.com/profefe/profefe
- https://ai.google/research/pubs/pub36575
- https://jvns.ca/blog/2017/09/24/profiling-go-with-pprof/
- https://github.com/google/pprof
- https://gianarb.it

Thanks

@gianarb / gianarb.it