Dependency Injection

A different way to structure a project

Whoami

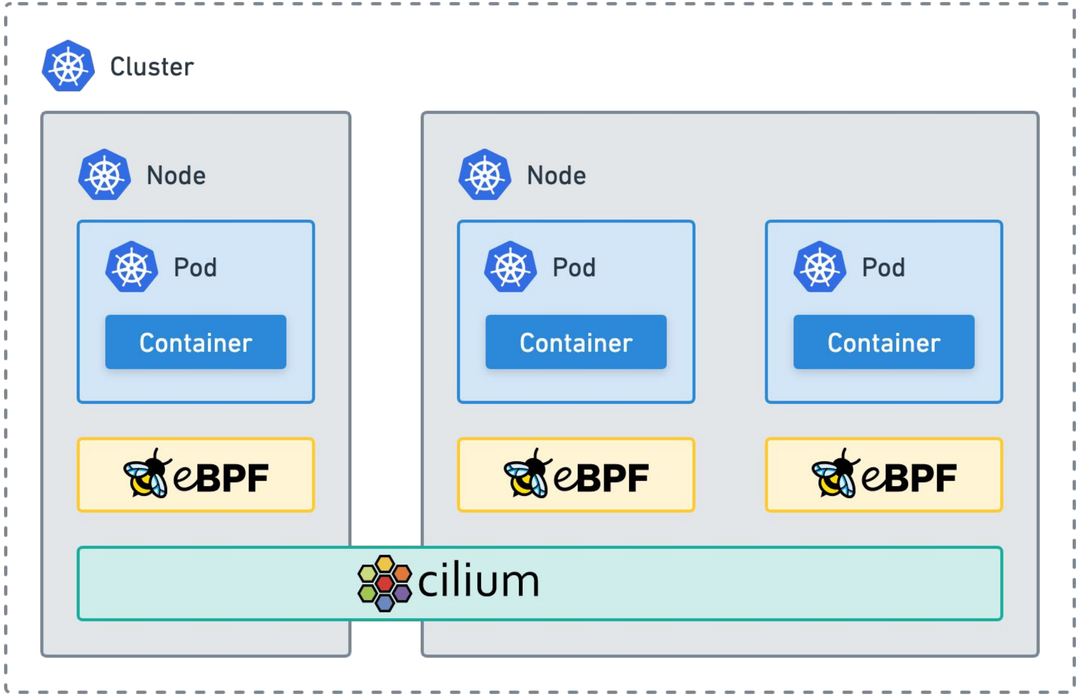
Dylan Reimerink

- Senior Software Engineer @ Isovalent
- On the Foundations & Loader team
- Cilium contributor
- https://github.com/dylandreimerink

The journey

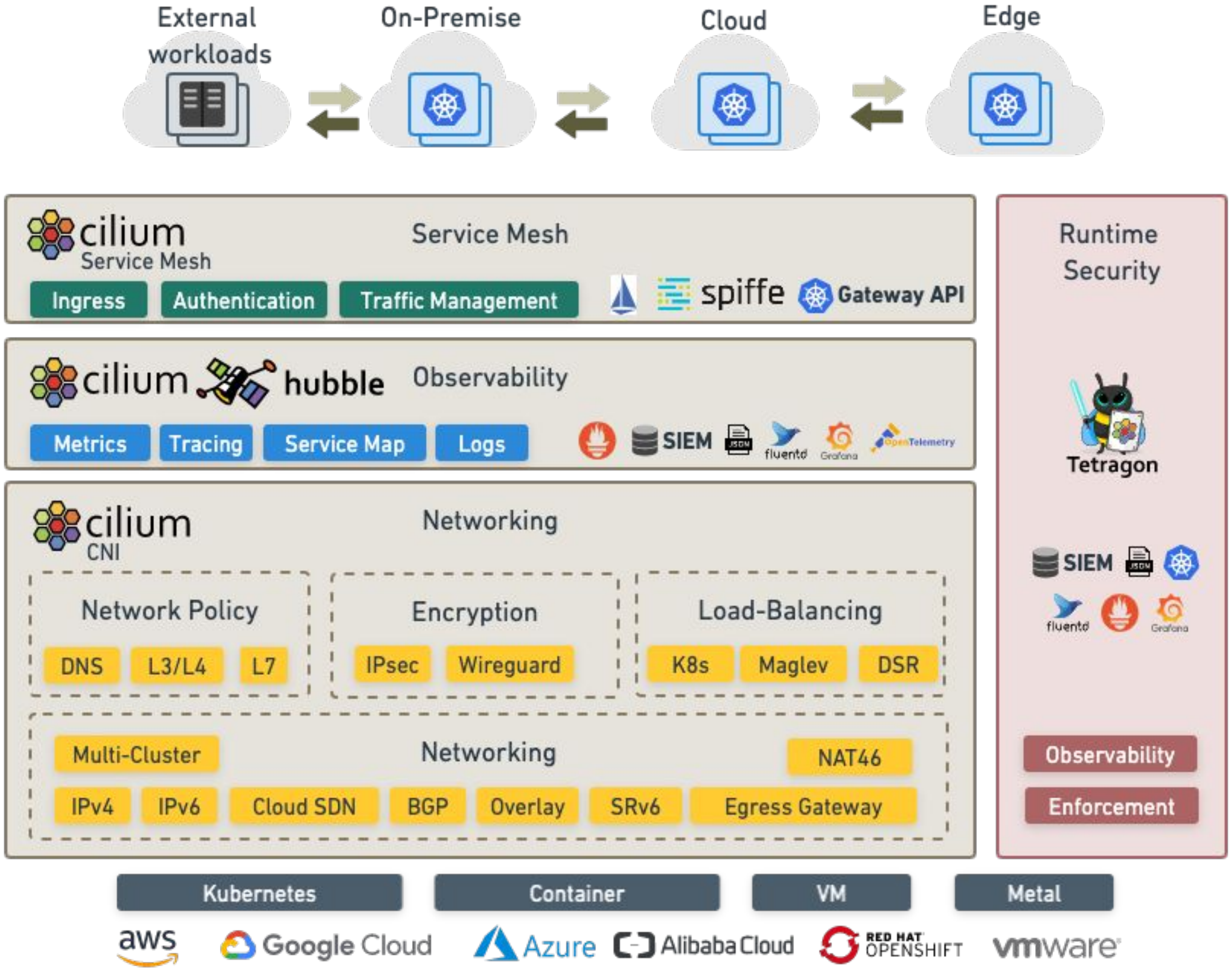
What is Cilium?

- eBPF base K8s networking
- Network security
- Network observability



What is Cilium?

- It does a lot
- 3rd most active project in CNCF



External workloads

On-Premise

Cloud

Edge

cilium Service Mesh

Service Mesh

Ingress

Authentication

Traffic Management

spiffe

Gateway API

cilium **hubble**

Observability

Metrics

Tracing

Service Map

Logs

SIEM

cilium CNI

Networking

Network Policy

DNS

L3/L4

L7

Encryption

IPsec

Wireguard

Load-Balancing

K8s

Maglev

DSR

Networking

Multi-Cluster

NAT46

IPv4

IPv6

Cloud SDN

BGP

Overlay

SRv6

Egress Gateway

Runtime Security

Tetragon

SIEM

Observability

Enforcement

Kubernetes

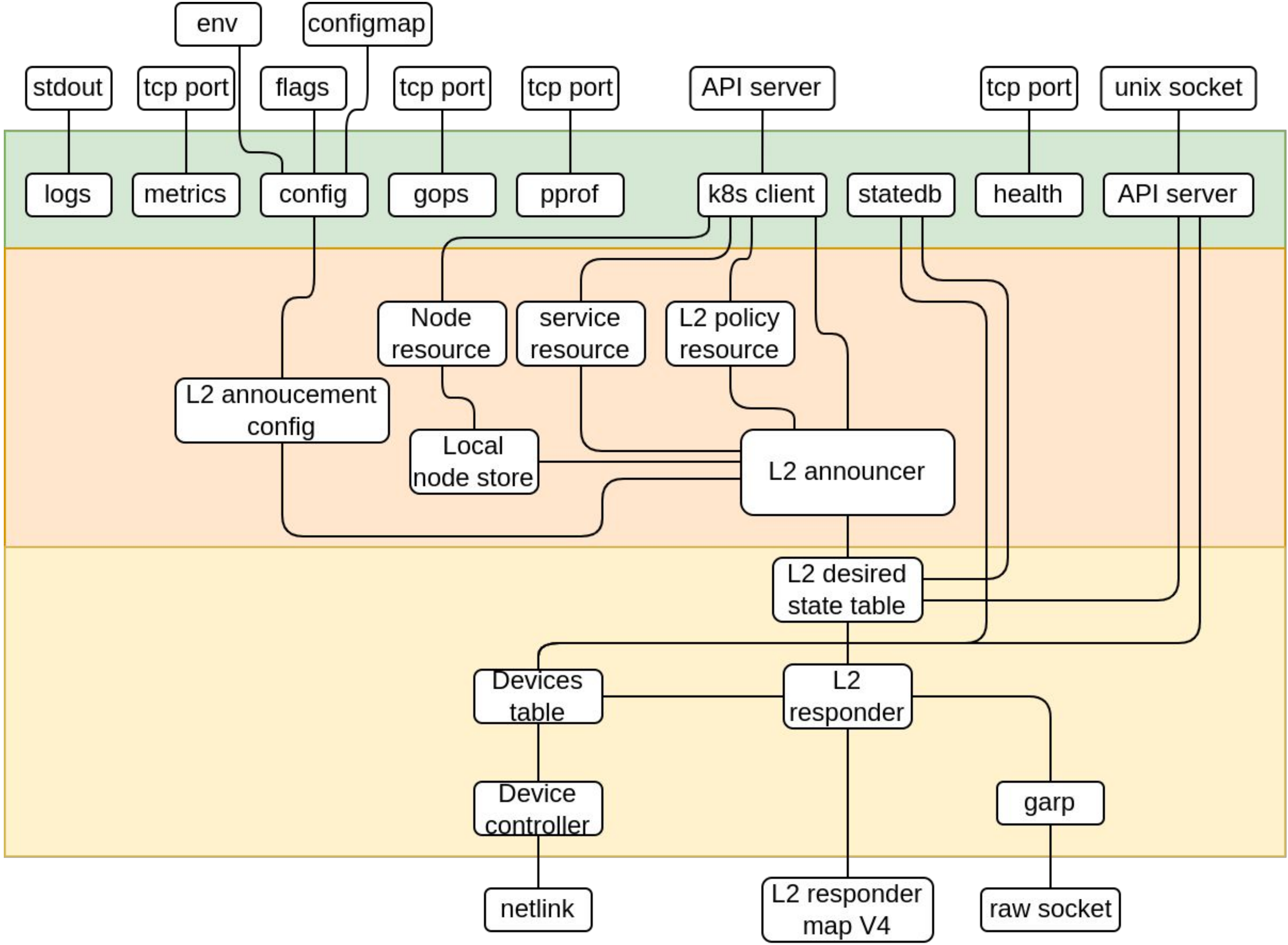
Container

VM

Metal

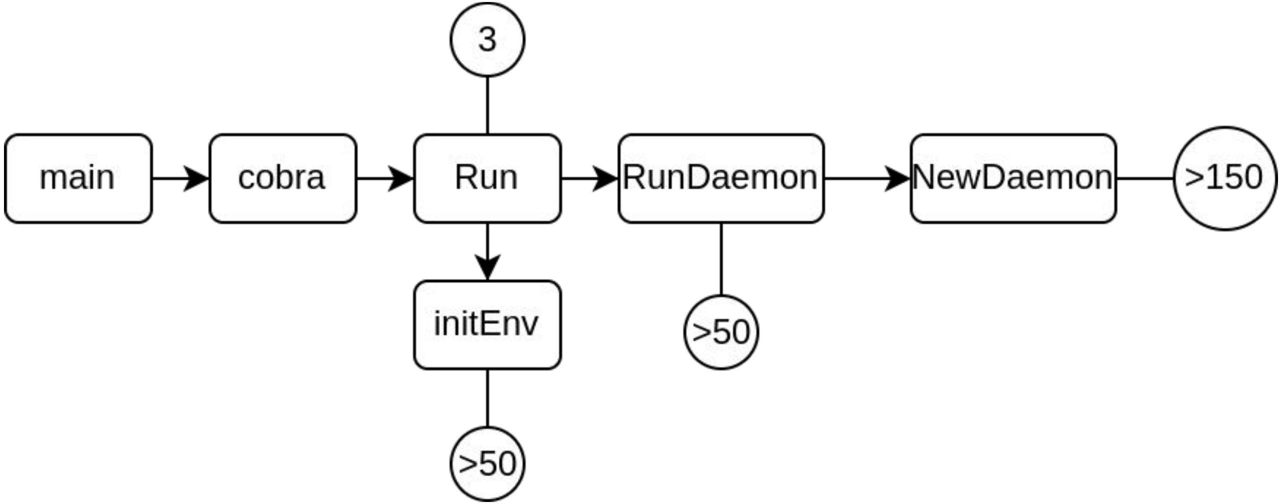
Components of a feature

- Infrastructure
- Control plane
- “Datapath”
- Kernel

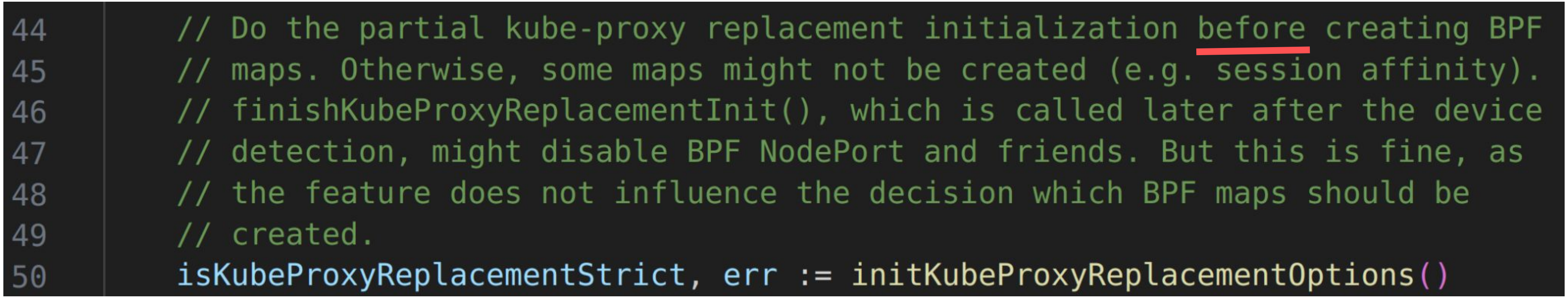


Cilium initialization

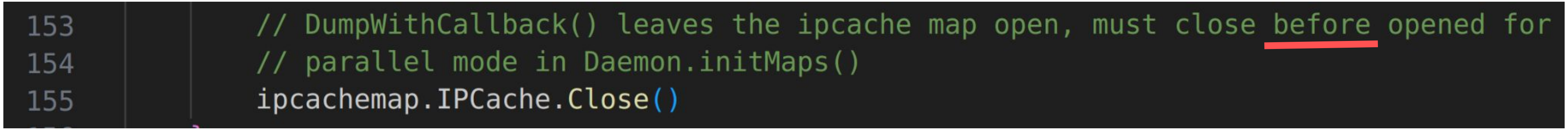
- Went back to v1.11, before we started our journey
- Initialization split over numerous call depths
- The largest “init” function `NewDaemon` being 733 lines
- `Daemon` as hub for accessing other components



Cilium initialization

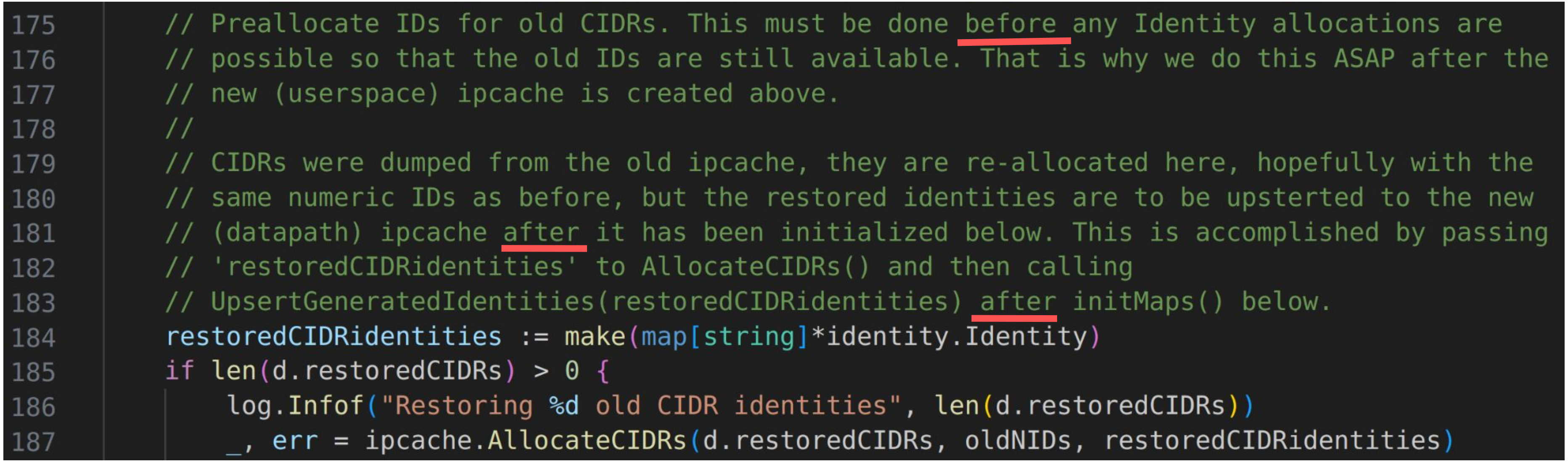


```
	// Do the partial kube-proxy replacement initialization before creating BPF
	// maps. Otherwise, some maps might not be created (e.g. session affinity).
	// finishKubeProxyReplacementInit(), which is called later after the device
	// detection, might disable BPF NodePort and friends. But this is fine, as
	// the feature does not influence the decision which BPF maps should be
	// created.
	isKubeProxyReplacementStrict, err := initKubeProxyReplacementOptions()
```

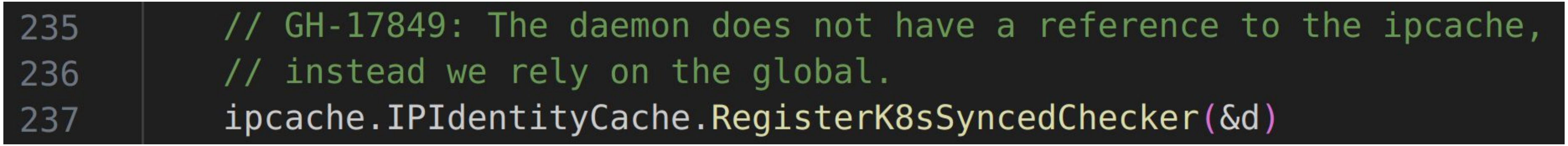


```
		// DumpWithCallback() leaves the ipcache map open, must close before opened for
		// parallel mode in Daemon.initMaps()
		ipcachemap.IPCache.Close()
```

Cilium initialization



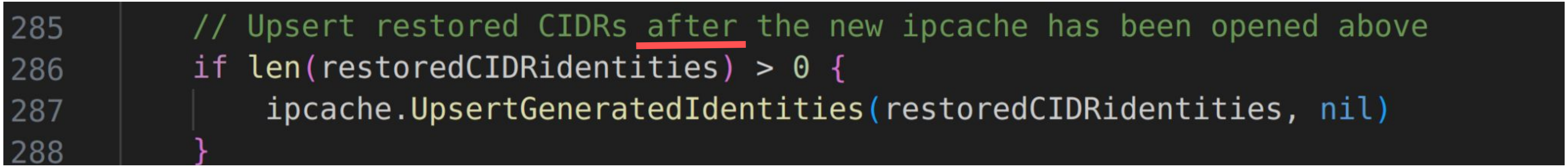
```go
	// Preallocate IDs for old CIDRs. This must be done before any Identity allocations are
	// possible so that the old IDs are still available. That is why we do this ASAP after the
	// new (userspace) ipcache is created above.
	//
	// CIDRs were dumped from the old ipcache, they are re-allocated here, hopefully with the
	// same numeric IDs as before, but the restored identities are to be upsterted to the new
	// (datapath) ipcache after it has been initialized below. This is accomplished by passing
	// 'restoredCIDRidentities' to AllocateCIDRs() and then calling
	// UpsertGeneratedIdentities(restoredCIDRidentities) after initMaps() below.
	restoredCIDRidentities := make(map[string]*identity.Identity)
	if len(d.restoredCIDRs) > 0 {
		log.Infof("Restoring %d old CIDR identities", len(d.restoredCIDRs))
		_, err = ipcache.AllocateCIDRs(d.restoredCIDRs, oldNIDs, restoredCIDRidentities)
```



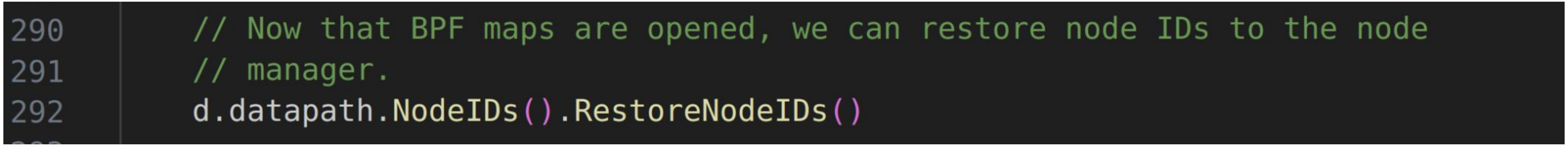
```go
	// GH-17849: The daemon does not have a reference to the ipcache,
	// instead we rely on the global.
	ipcache.IPIdentityCache.RegisterK8sSyncedChecker(&d)
```

Cilium initialization

```go
// watchers.NewCiliumNodeUpdater needs to be registered *after* d.endpointManager
d.k8sWatcher.NodeChain.Register(watchers.NewCiliumNodeUpdater(d.nodeDiscovery))
```

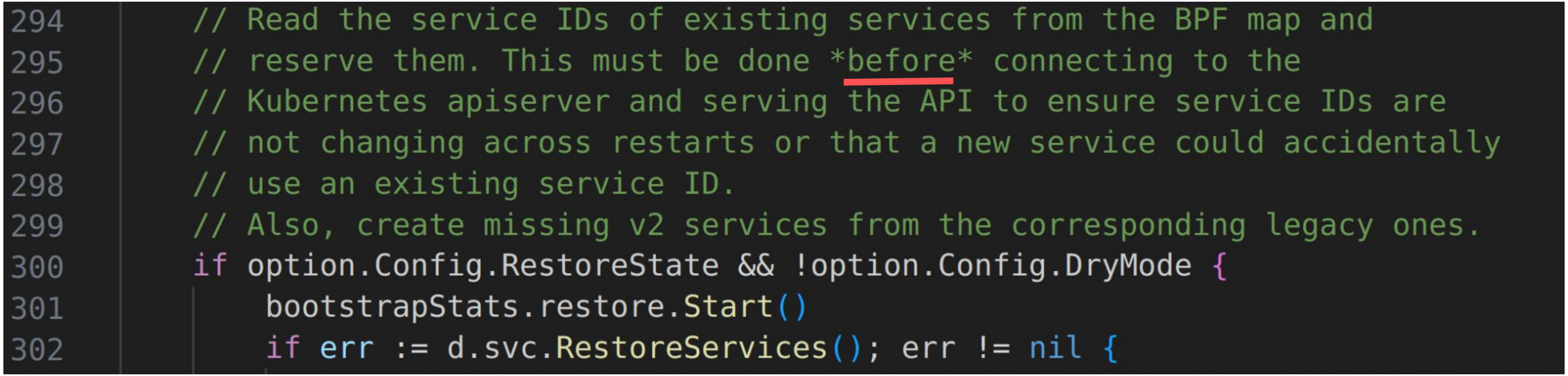


```go
// Upsert restored CIDRs after the new ipcache has been opened above
if len(restoredCIDRidentities) > 0 {
	ipcache.UpsertGeneratedIdentities(restoredCIDRidentities, nil)
}
```



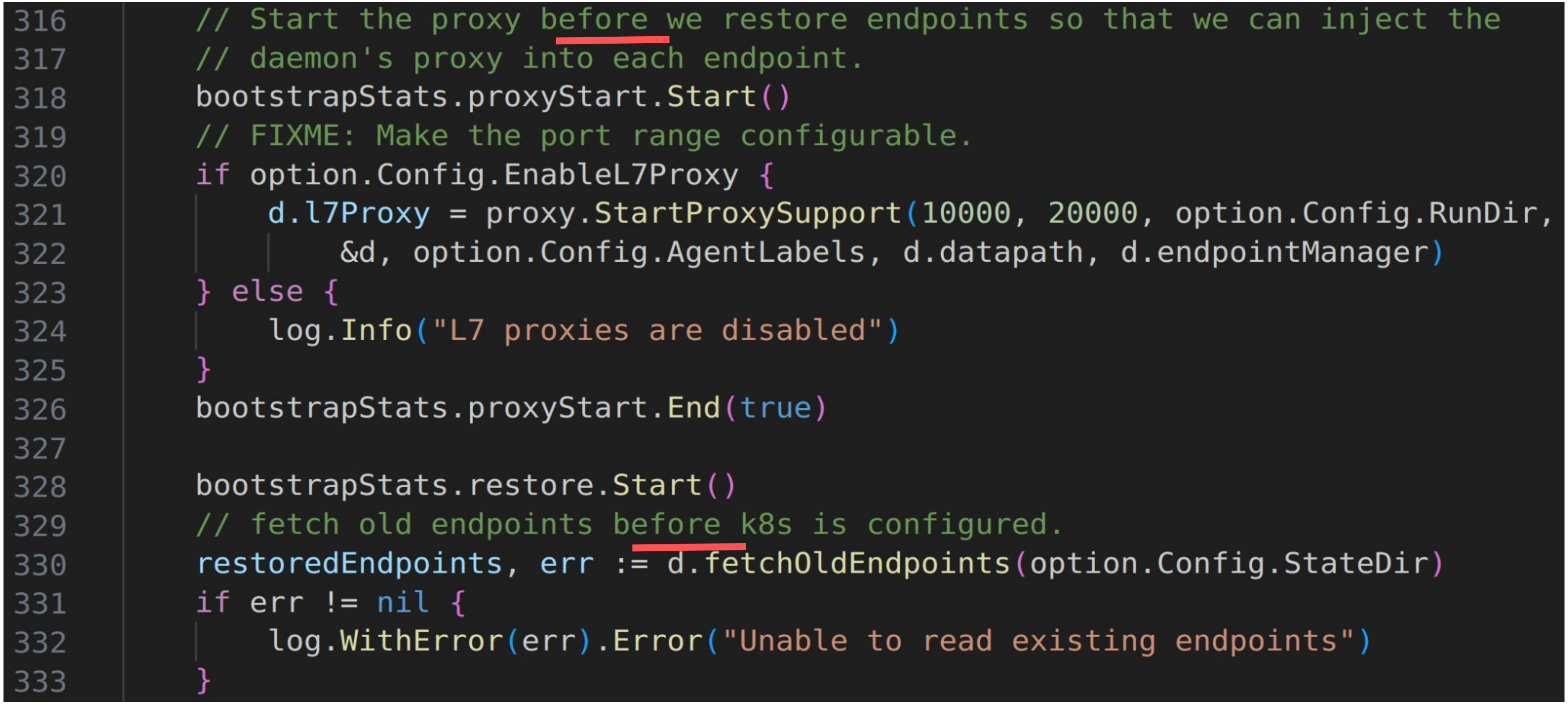
```go
// Now that BPF maps are opened, we can restore node IDs to the node
// manager.
d.datapath.NodeIDs().RestoreNodeIDs()
```

Cilium initialization



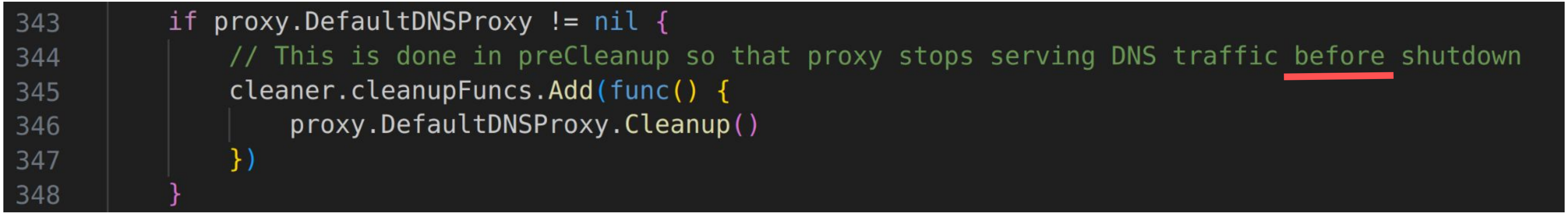
```
    // Read the service IDs of existing services from the BPF map and
    // reserve them. This must be done *before* connecting to the
    // Kubernetes apiserver and serving the API to ensure service IDs are
    // not changing across restarts or that a new service could accidentally
    // use an existing service ID.
    // Also, create missing v2 services from the corresponding legacy ones.
    if option.Config.RestoreState && !option.Config.DryMode {
        bootstrapStats.restore.Start()
        if err := d.svc.RestoreServices(); err != nil {
```

Cilium initialization



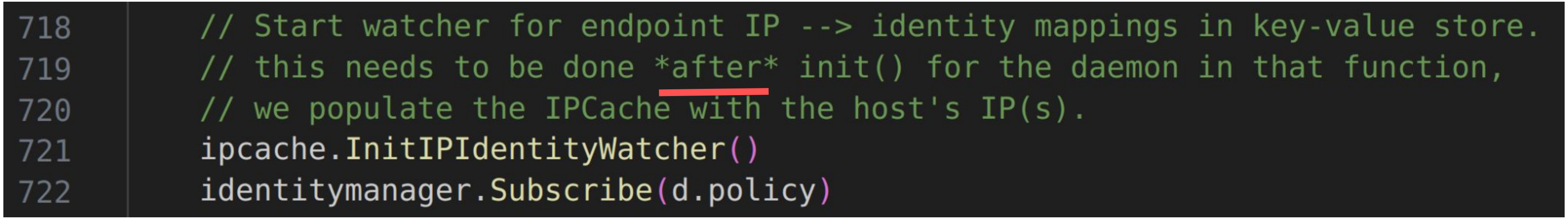
```
    // Start the proxy before we restore endpoints so that we can inject the
    // daemon's proxy into each endpoint.
    bootstrapStats.proxyStart.Start()
    // FIXME: Make the port range configurable.
    if option.Config.EnableL7Proxy {
        d.l7Proxy = proxy.StartProxySupport(10000, 20000, option.Config.RunDir,
            &d, option.Config.AgentLabels, d.datapath, d.endpointManager)
    } else {
        log.Info("L7 proxies are disabled")
    }
    bootstrapStats.proxyStart.End(true)

    bootstrapStats.restore.Start()
    // fetch old endpoints before k8s is configured.
    restoredEndpoints, err := d.fetchOldEndpoints(option.Config.StateDir)
    if err != nil {
        log.WithError(err).Error("Unable to read existing endpoints")
    }
```

Cilium initialization



```
343     if proxy.DefaultDNSProxy != nil {
344         // This is done in preCleanup so that proxy stops serving DNS traffic before shutdown
345         cleaner.cleanupFuncs.Add(func() {
346             proxy.DefaultDNSProxy.Cleanup()
347         })
348     }
```

I had to stop here, the last such comment was almost at the end at line 718



```
718     // Start watcher for endpoint IP --> identity mappings in key-value store.
719     // this needs to be done *after* init() for the daemon in that function,
720     // we populate the IPCache with the host's IP(s).
721     ipcache.InitIPIdentityWatcher()
722     identitymanager.Subscribe(d.policy)
```

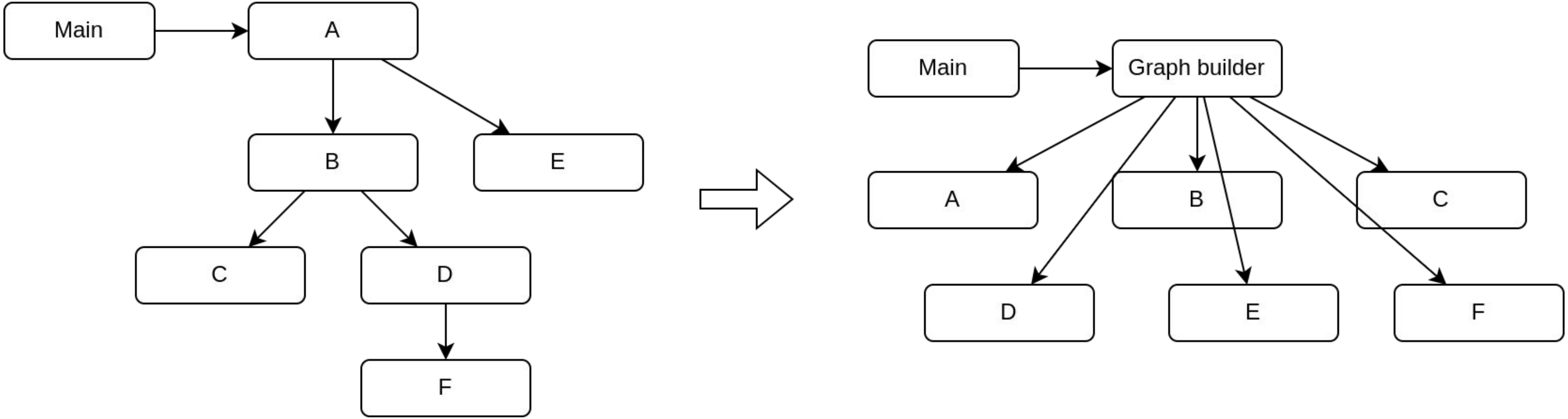
“The problem”

- Lots of nested dependencies (inherent)
- Implicit dependencies
    - Global variables
    - System state
- Hard to modify
- Hard to shutdown (correctly)
- Hard to test

Dependency injection

What is dependency injection?

- Declarative dependencies instead of imperative dependencies
- Popular in other languages/frameworks like Java, C#, PHP
- Taking all dependencies as arguments to a constructor



Introducing go.uber.org/fx

- Made and maintained by Uber
    - Originally developed by Glib (now a colleague of mine)
- Battle tested and maintained
- As is DI framework, use go.uber.org/dig if you want to customize
- Made to solve almost the same problems we were having

Provides and invokes

```go
var logger = slog.New(slog.NewTextHandler(os.Stdout, &slog.HandlerOptions{}))

func main() {
   listener,   := net.Listen("tcp", ":8080")
   server := Server{
       logger:   logger,
       listener: listener,
   }
   _ = server.Serve()
}

type Server struct {
   http.Server
   logger   *slog.Logger
   listener net.Listener
}

func (s *Server) Serve() error {
   /* [...] */
   return nil
}
```

Provides and invokes

```go
func main() {
    app := fx.New(
        fx.Provide(newListener),
        fx.Provide(newLogger),
        fx.Invoke(NewServer),
    )
    app.Run()
}

func newListener() (net.Listener, error) {
    return net.Listen("tcp", ":8080")
}

func newLogger() *slog.Logger {
    return slog.New(
        slog.NewTextHandler(
            os.Stdout,
            &slog.HandlerOptions{},
        ))
}
```

```go
type Server struct {
    logger   *slog.Logger
    listener net.Listener
}

func NewServer(
    logger *slog.Logger,
    listener net.Listener,
) *Server {
    return &Server{
        logger:   logger,
        listener: listener,
    }
}

func (s *Server) Serve() error {
    /* [...] */
    return nil
}
```

Lifecycles

```go
type Server struct {
    http.Server
    logger   *slog.Logger
    listener net.Listener
}

func NewServer(
    lifecycle fx.Lifecycle,
    logger *slog.Logger,
    listener net.Listener,
) *Server {
    s := &Server{
        logger:   logger,
        listener: listener,
    }
    lifecycle.Append(fx.Hook{
        OnStart: s.OnStart,
        OnStop:  s.OnStop,
    })
    return s
}
```

```go
func (s *Server) OnStart(_ context.Context) error {
    go s.Serve()
    return nil
}

func (s *Server) OnStop(ctx context.Context) error {
    return s.Shutdown(ctx)
}

func (s *Server) Serve() {
    s.Server.Handler = http.DefaultServeMux
    _ = s.Server.Serve(s.listener)
}
```

Lifecycles

```go
func main() {
   app := fx.New(
       fx.NopLogger,
       fx.Provide(NewA),
       fx.Provide(NewB),
       fx.Invoke(NewC),
   )
   app.Run()
}

func NewA(lifecycle fx.Lifecycle) *A {
   print(lifecycle, "A")
   return &A{}
}

func NewB(lifecycle fx.Lifecycle, a *A) *B {
   print(lifecycle, "B")
   return &B{}
}
```

```go
func NewC(lifecycle fx.Lifecycle, b *B) *C {
   print(lifecycle, "C")
   return &C{}
}

func print(lifecycle fx.Lifecycle, name string) {
   fmt.Println(name + " constructor")
   lifecycle.Append(fx.Hook{
       OnStart: func(context.Context) error {
           fmt.Println(name + " start")
           return nil
       },
       OnStop: func(context.Context) error {
           fmt.Println(name + " stop")
           return nil
       },
   })
}
```

Lifecycles

```
func main() {
   app := fx.New(
       fx.NopLogger,
       fx.Provide(NewA),
       fx.Provide(NewB),
       fx.Invoke(NewC),
   )
   app.Run()
}

func NewA(lifecycle fx.Lif
   print(lifecycle, "A")
   return &A{}
}

func NewB(lifecycle fx.Lif
   print(lifecycle, "B")
   return &B{}
}
```

```
$ go run .
A constructor
B constructor
C constructor
A start
B start
C start
^C
C stop
B stop
A stop
```

```
fecycle, b *B) *C {




ifecycle, name string) {
onstructor")
ok{
ext.Context) error {
me + " start")


xt.Context) error {
me + " stop")
```

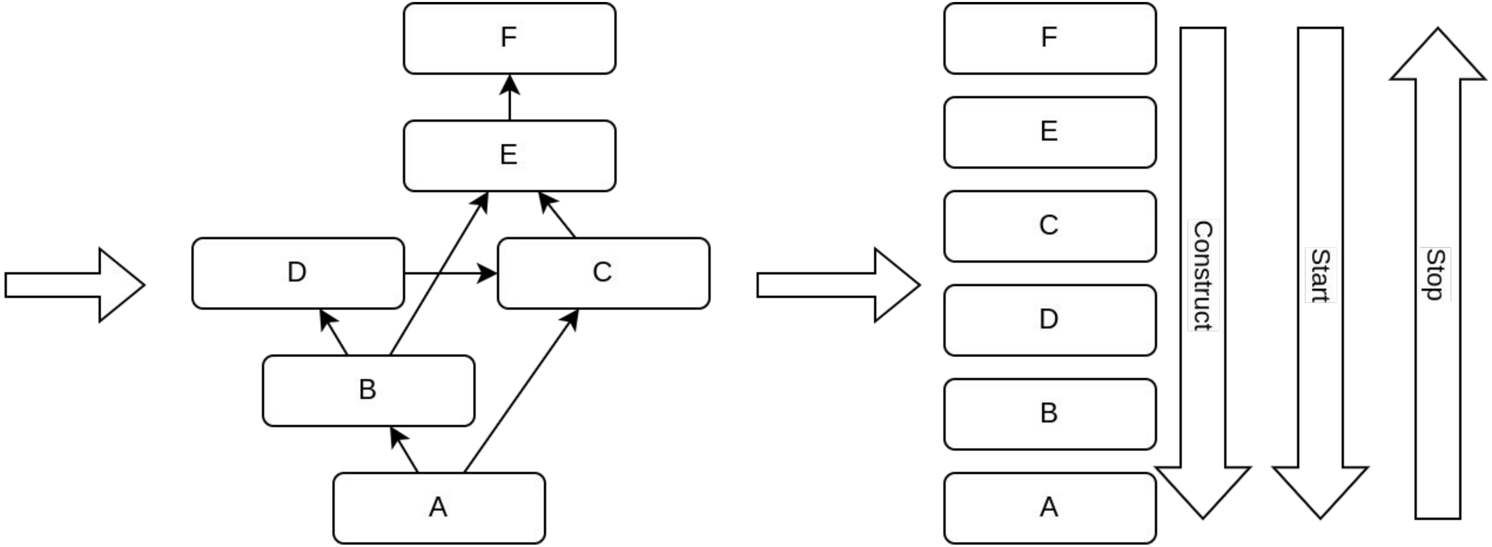
Groups

```
type MuxParams struct {
   fx.In
   Handles []MuxHandle `group:"mux_handles"`
}

func NewMux(params MuxParams) *http.ServeMux {
   mux := http.NewServeMux()
   for  , handle := range params.Handles {
       mux.HandleFunc(handle.path, handle.handler)
   }
   return mux
}

type FooOut struct {
   fx.Out
   Handle MuxHandle `group:"mux_handles"`
}
```

```
type BarOut struct {
   fx.Out
   Handle MuxHandle `group:"mux_handles"`
}

func FooHandler() FooOut {
   return FooOut{
       Handle: MuxHandle{
           path:    "/foo",
           handler: /* ... */,
       },
   }
}

func BarHandler() BarOut {
   return BarOut{
       Handle: MuxHandle{
           path:    "/bar",
           handler: /* ... */,
       },
   }
}
```

Under the hood

```
func A(B, C) A
func B(D, E) B
func C(E) C
func D(C) D
func E(F) F
func F() F
```



Tips, tricks and lessons learned

Inject, but in moderation

- Not everything has to be a component
- Rule of thumb, only inject stateful components
- For example
  - Logger Yes
  - Math lib No
  - Config Yes
  - Strings lib No

Pick logical boundaries

- A component can internally be complex, not every type has to go in the graph
- Packages are excellent boundaries, leverage exported vs unexported types
- Components can be bundled with fx.Options to make the a hierarchy resembling the package structure

Provide targeted interfaces

- Using interfaces makes it easy to swap out a component
- Small interfaces are easy to mock/fake in tests
- Using interfaces makes it impossible for external components to rely on internal implementation
- A single value can be exported as multiple interfaces

Go easy on the groups

- Groups are great if multiple consumers might be interested in an array of values
    - Metrics, Config, HTTP endpoints
    - Collected by a registry/server but also by tools to generate docs for example
- The connection is less explicit than with Provide/Invoke

Stay with static graphs when possible

- It is tempting to apply conditional logic to constructing your graph
    - Only add component X if flag Y is set
- Dynamic graphs are had to validate
- Instead internally disable a component or conditionally hook into the lifecycle

Always provide “safe” values

- Returning nil from a constructor will cause panics at some point
- Let a component be disabled under certain configuration
  - Then provide a IsDisabled() bool method
  - Or making methods failable
  - Both make it explicit that the component may fail or be unavailable

Thank you, Questions?