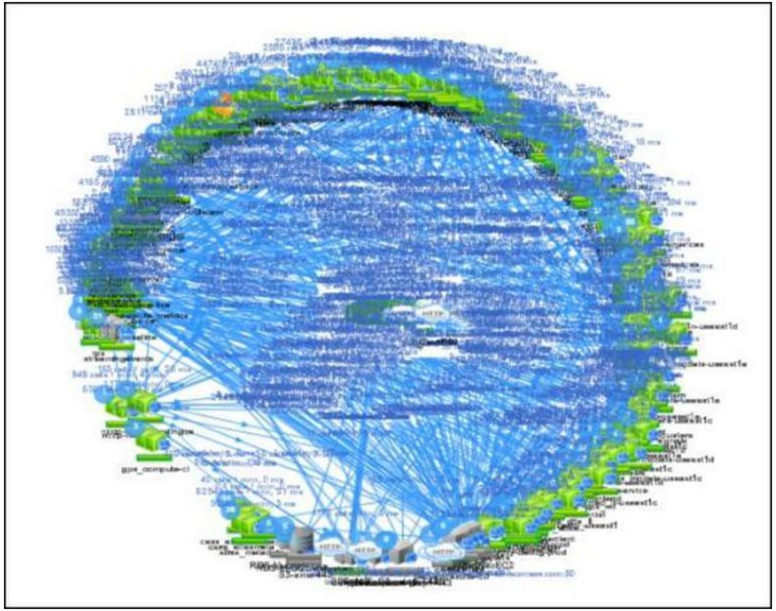
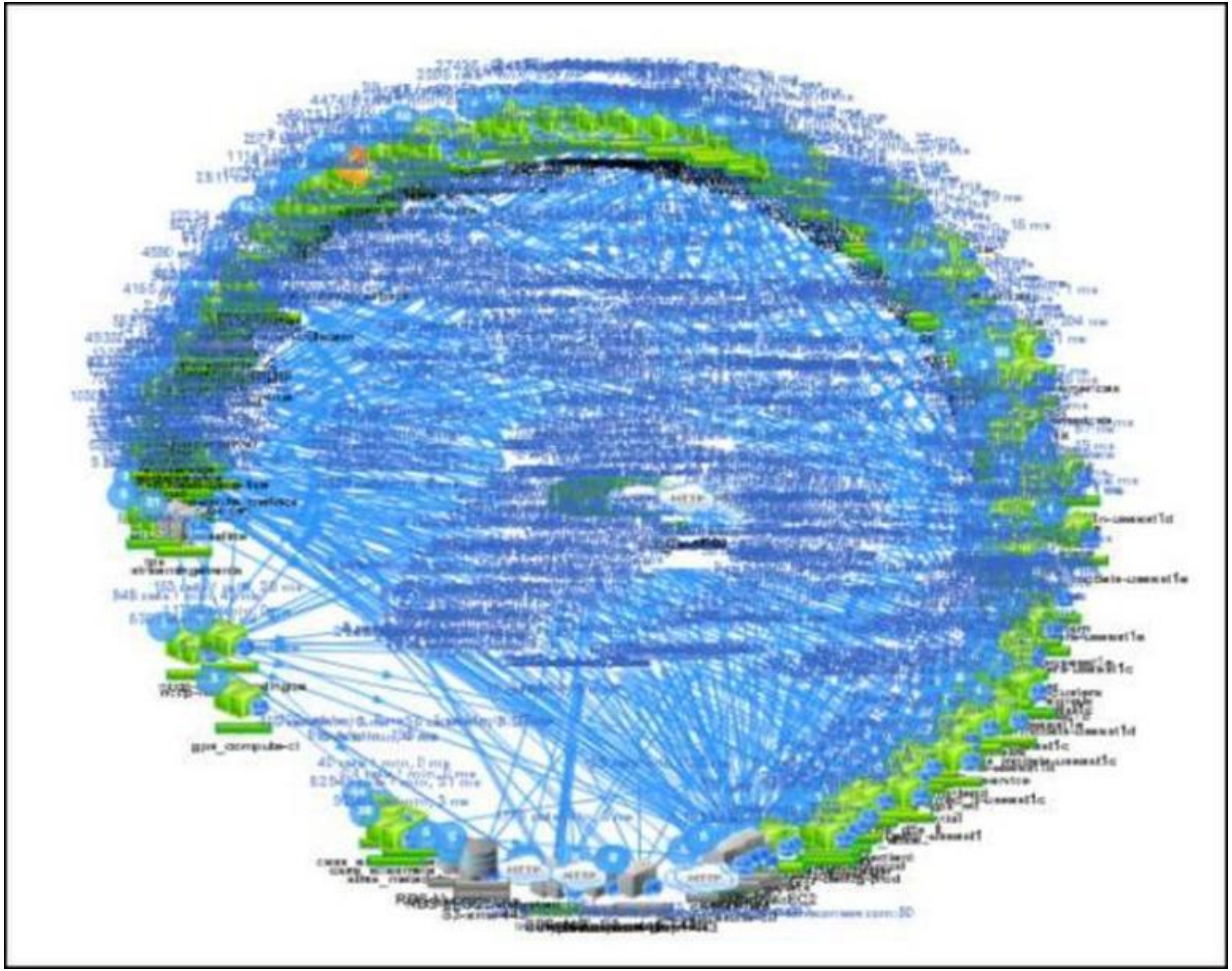
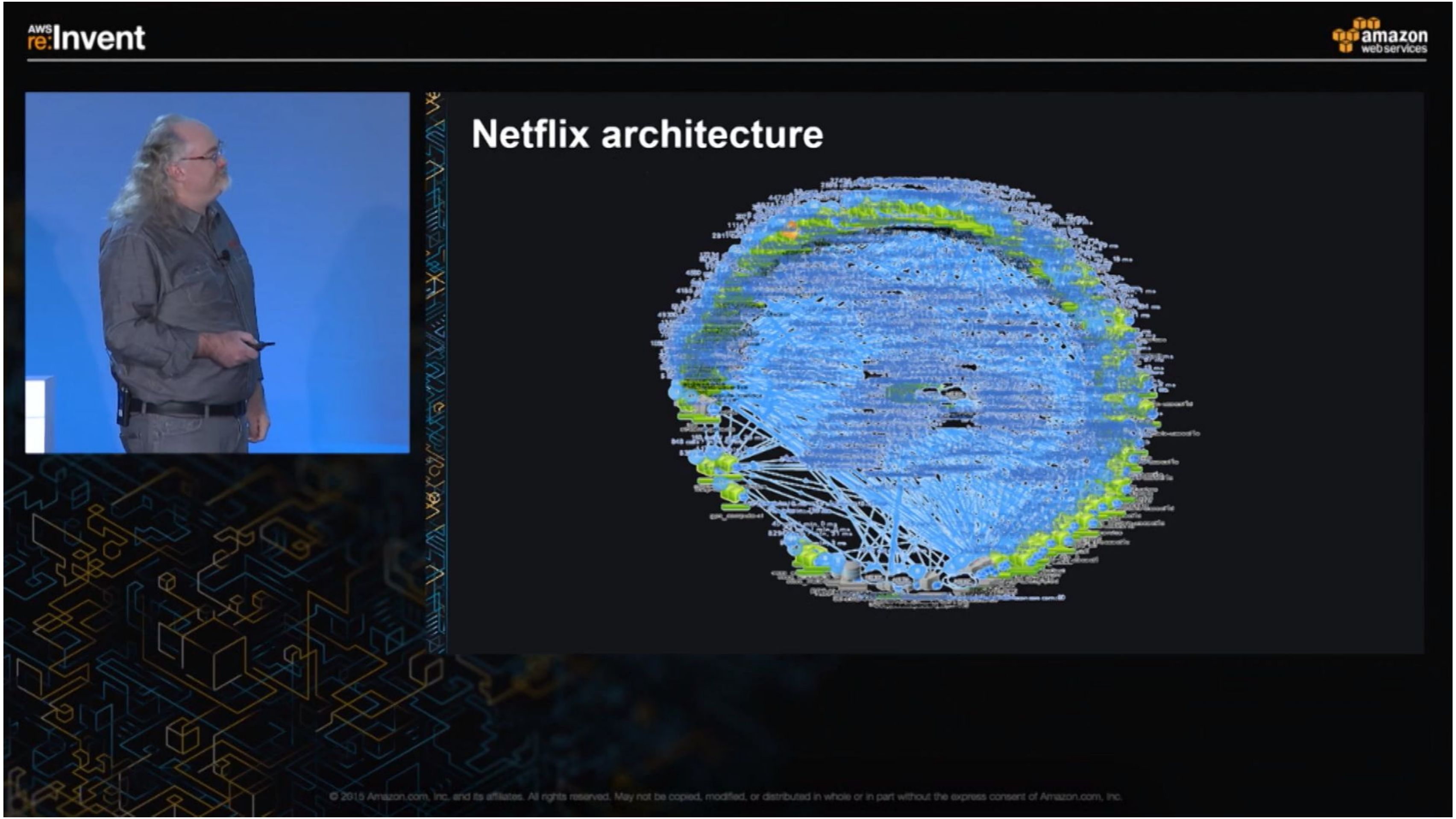
Microservices







Netflix architecture

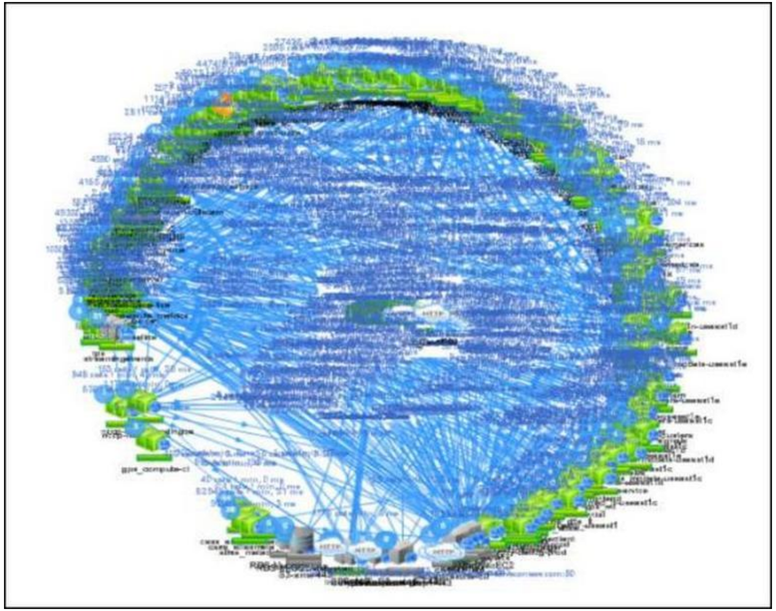
# whoami

- Martin Štefanko
- Software engineer, Red Hat
- MicroProfile committer
- Microservices enthusiast
- 🐦@xstefank

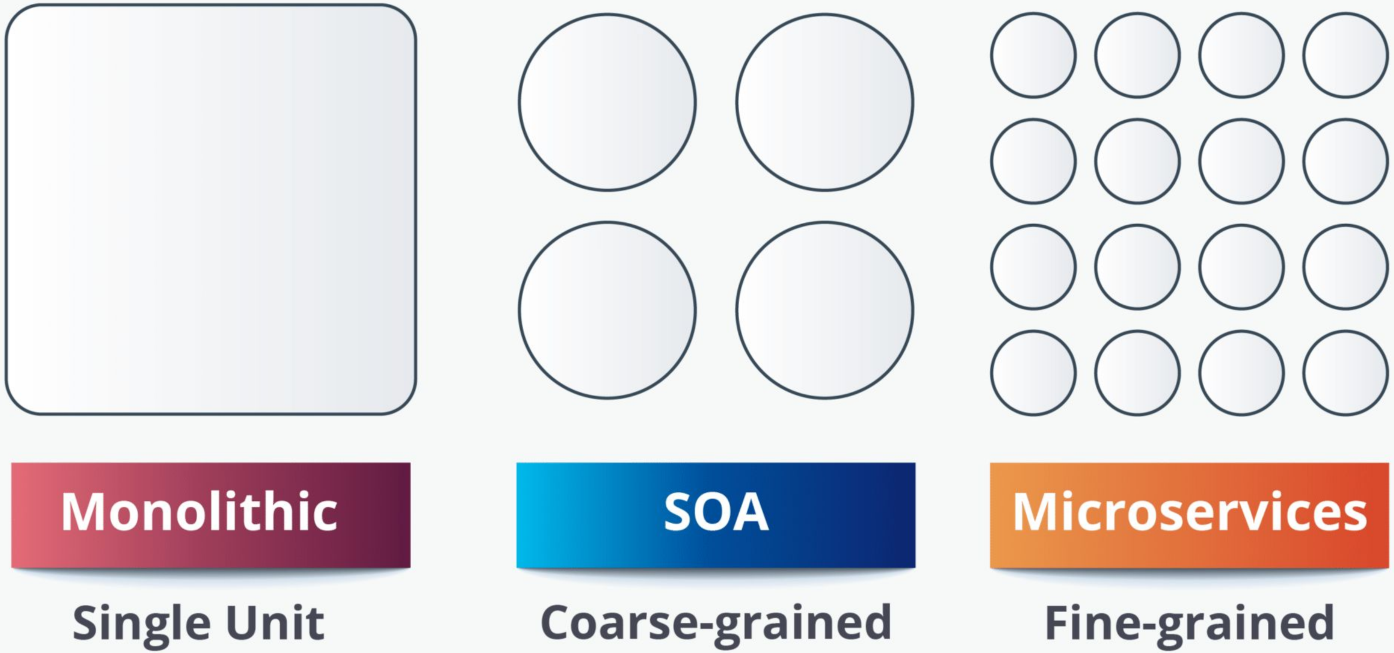
What we are going to go through?

- **1st lecture** - Microservices introduction
- **2nd lecture** - Introduction to Quarkus, HTTP, REST, Docker/Podman, and Kubernetes/Openshift
- **3rd lecture** - MicroProfile, Jakarta EE
- **4th lecture** - MicroProfile continued, Panache, reactive programming, (Kafka)

Microservices



Monolithic vs. SOA vs. Microservices



Monolithic

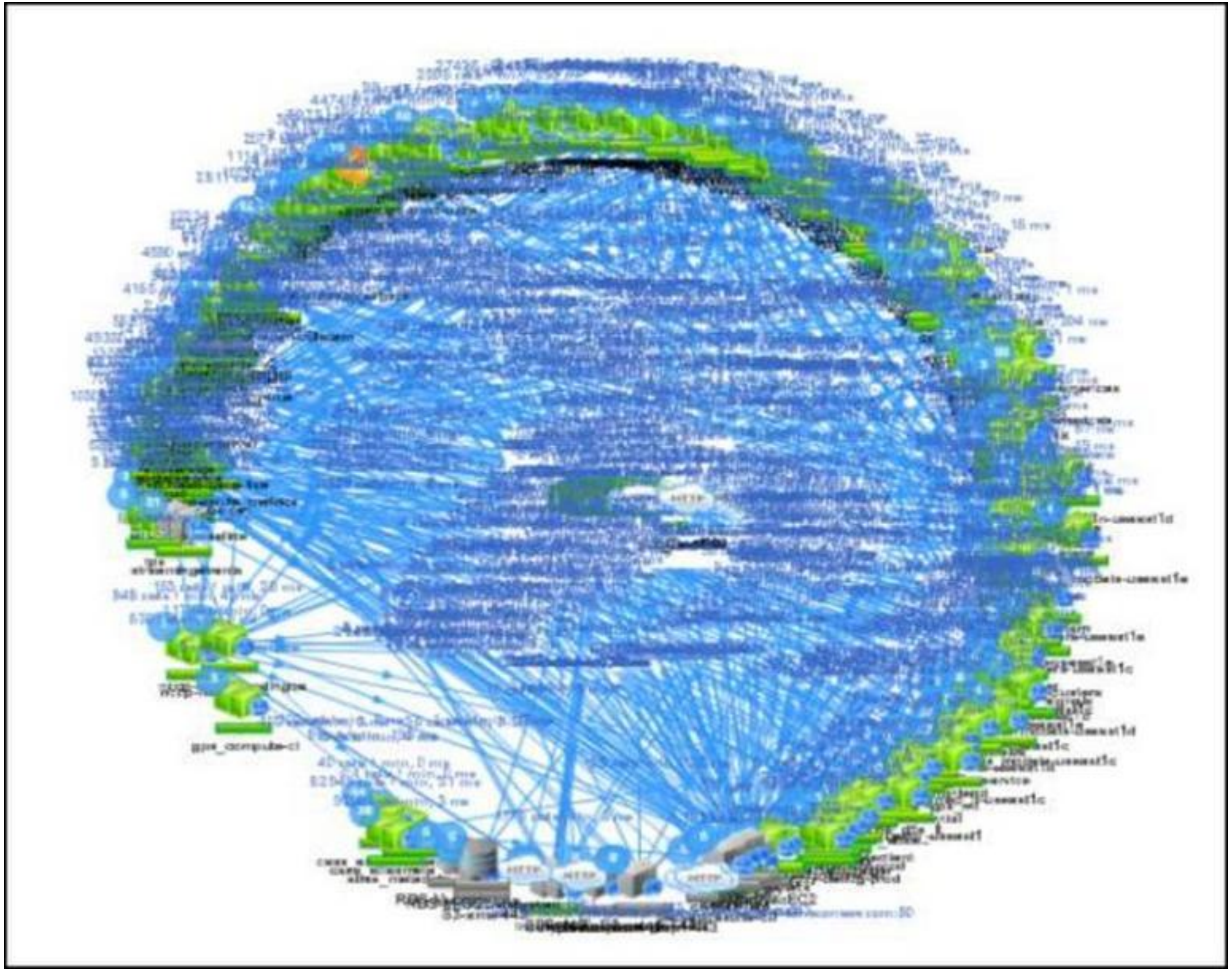
Single Unit

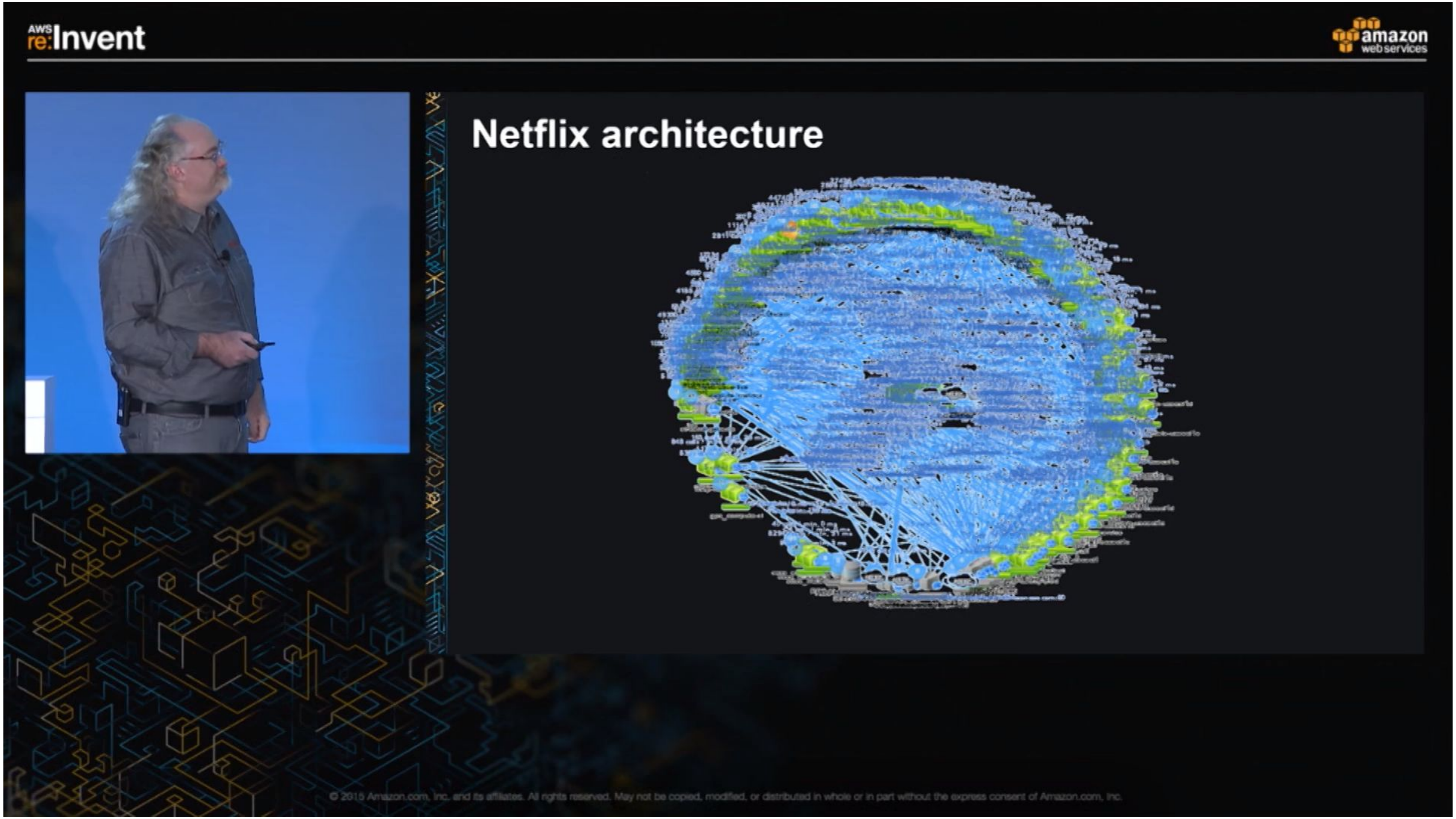
SOA

Coarse-grained

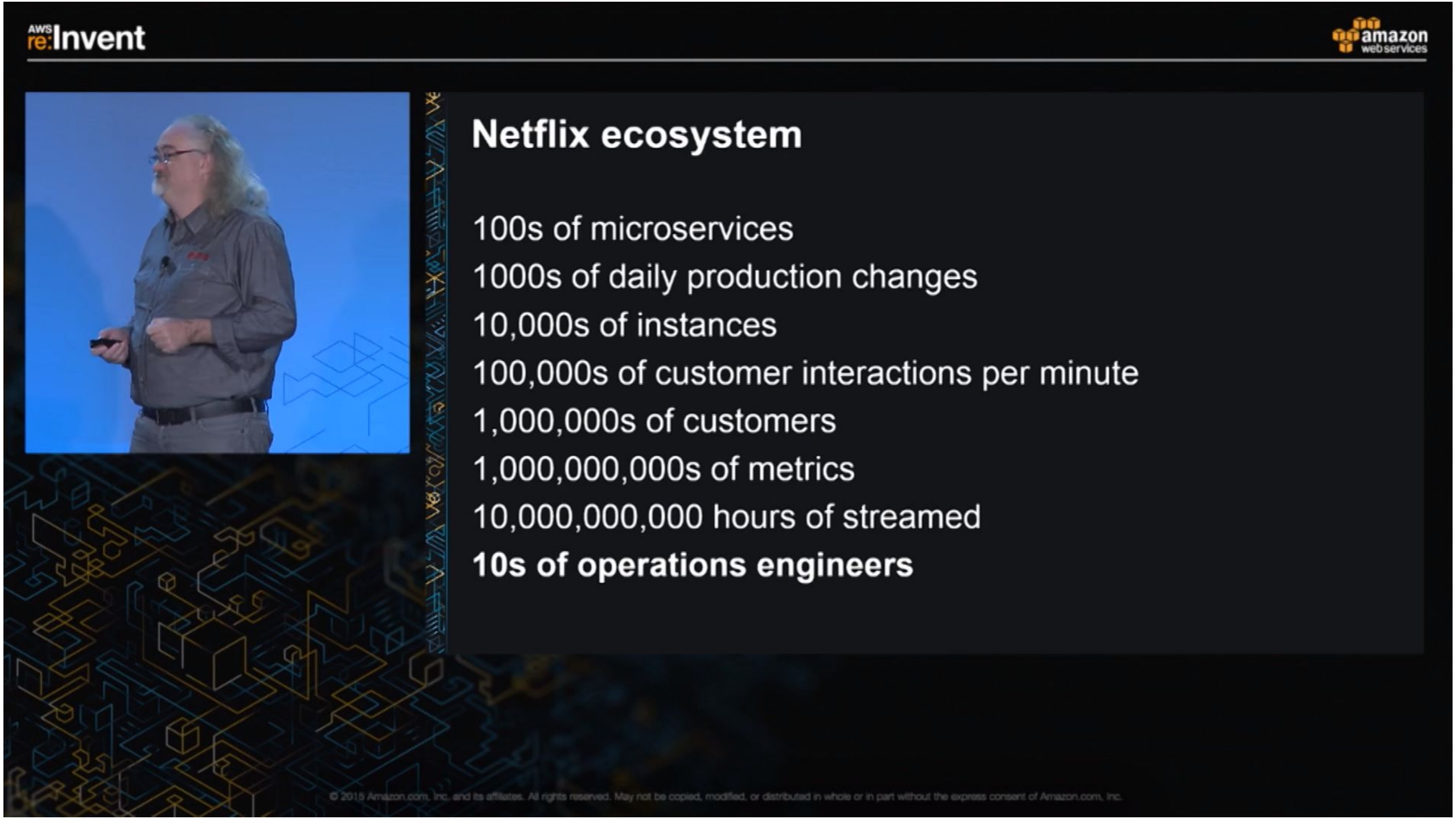
Microservices

Fine-grained





Netflix architecture



Netflix ecosystem

100s of microservices

1000s of daily production changes

10,000s of instances

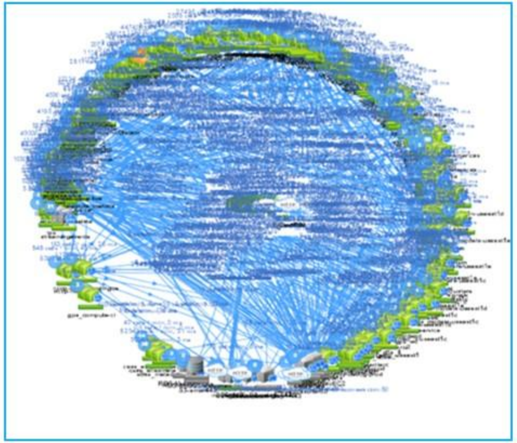
100,000s of customer interactions per minute

1,000,000s of customers

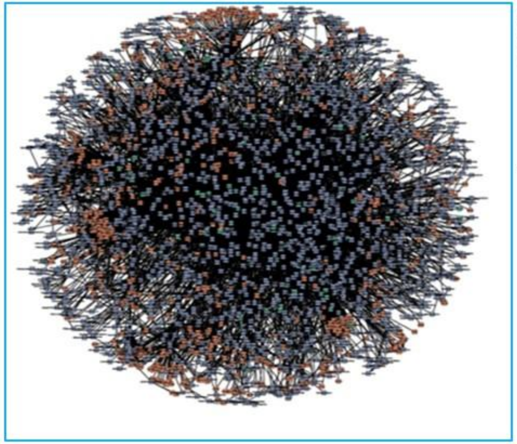
1,000,000,000s of metrics

10,000,000,000 hours of streamed

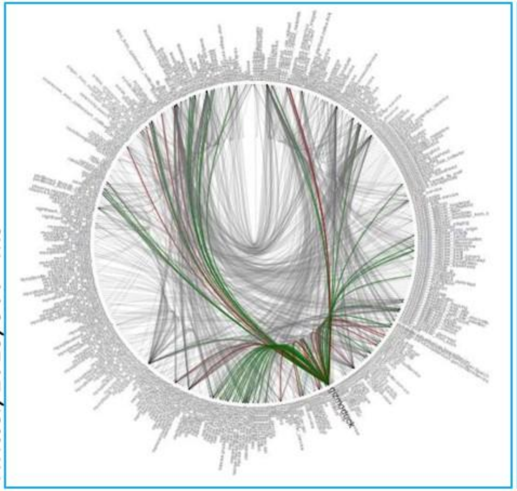
10s of operations engineers



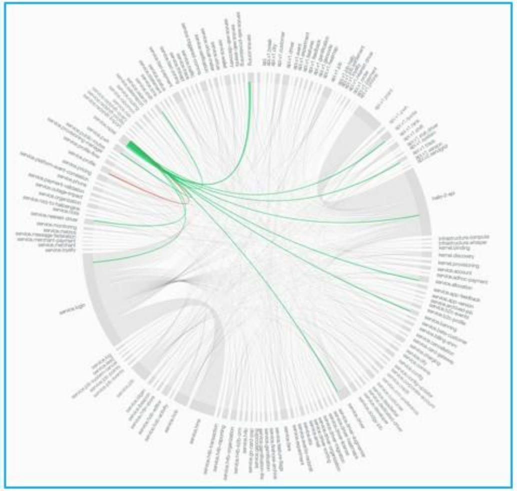
Netflix, 500+ ms



Amazon, 2009



Twitter, 2013, 500+ ms



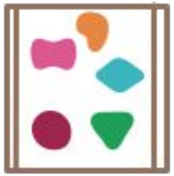
Hailo, 450+ ms

Monolith

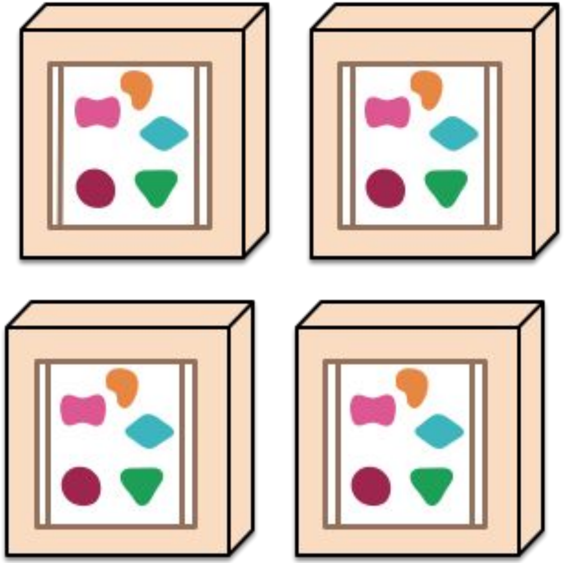
Monolithic applications

- Common development model (past 20+ years)
- Single or small amount of deployments
- Application server
- Single process
- Components tightly coupled

A monolithic application puts all its functionality into a single process...



... and scales by replicating the monolith on multiple servers



Monolith - advantages

- Development model == application requirements
  - Traditionally CRUD or MVC
  - Presentation - Business layer - Database
- Single or small number of archives/deployments
- Easy horizontal scaling

Monolith - problems

- Adding new functionality
- General maintenance (bug fixing, CVEs)
- Every single change means requires rebuild and redeploy of the whole application
- Replicated server instances take more resources
- Slow startup times

Monolith - problems

- Often extremely large code bases
    - Hard to understand / maintain
    - Long learning curves
    - Experts in particular system / part of the system
    - Often a commitment to a particular technology
        - Or even a specific version

Microservices

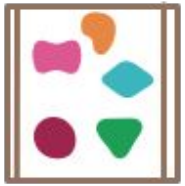
Architectural pattern

- System as a collection of small, isolated services
- Each service
  - Owns its data
  - Is independently isolated
  - Is scalable
  - Is resilient
  - Is self-maintained
- Evolution of SOA
- Intuitive approach

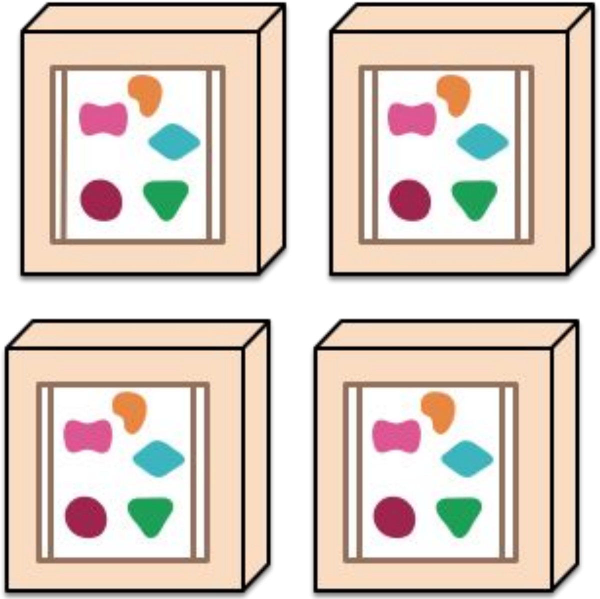
Architectural pattern

- Each service represents the separated and independent part of the system
- Interaction with other microservices is allowed only through predefined communication interfaces (API)

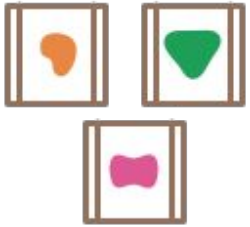
A monolithic application puts all its functionality into a single process...



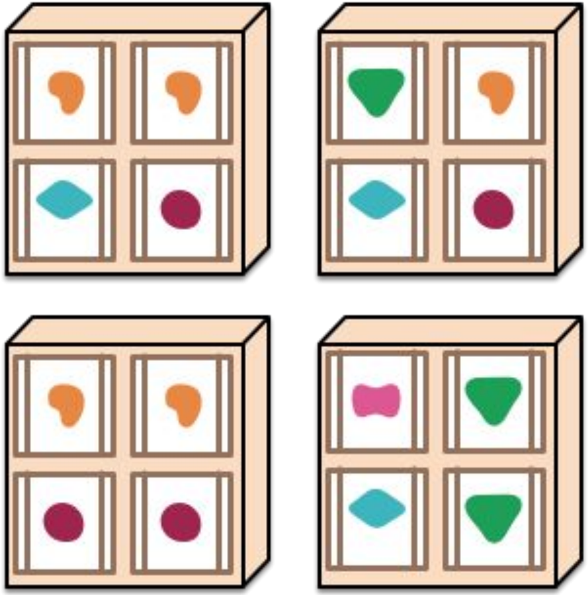
... and scales by replicating the monolith on multiple servers



A microservices architecture puts each element of functionality into a separate service...



... and scales by distributing these services across servers, replicating as needed.



Organization structure

- Teams segregation according to the architecture
- Each team is responsible for one or a small set of services
- Structured according to the business goals
  - Teams should be self-maintained
  - Devs, Testers, Front-end, ...
- One team should not have access/knowledge of different services/teams

Isolation

- Ownership of resources
- Data requests to different services prohibited only through the API
  - Control the access, computation requirements
- Service must act as an external component
- Loosely coupled services
- Virtual addresses
- Scaling, load-balancing
- Resiliency

Isolation

- Underlying technology may differ
  - Different languages
  - Different runtimes
  - Different frameworks
  - Different versions

Law of Demeter

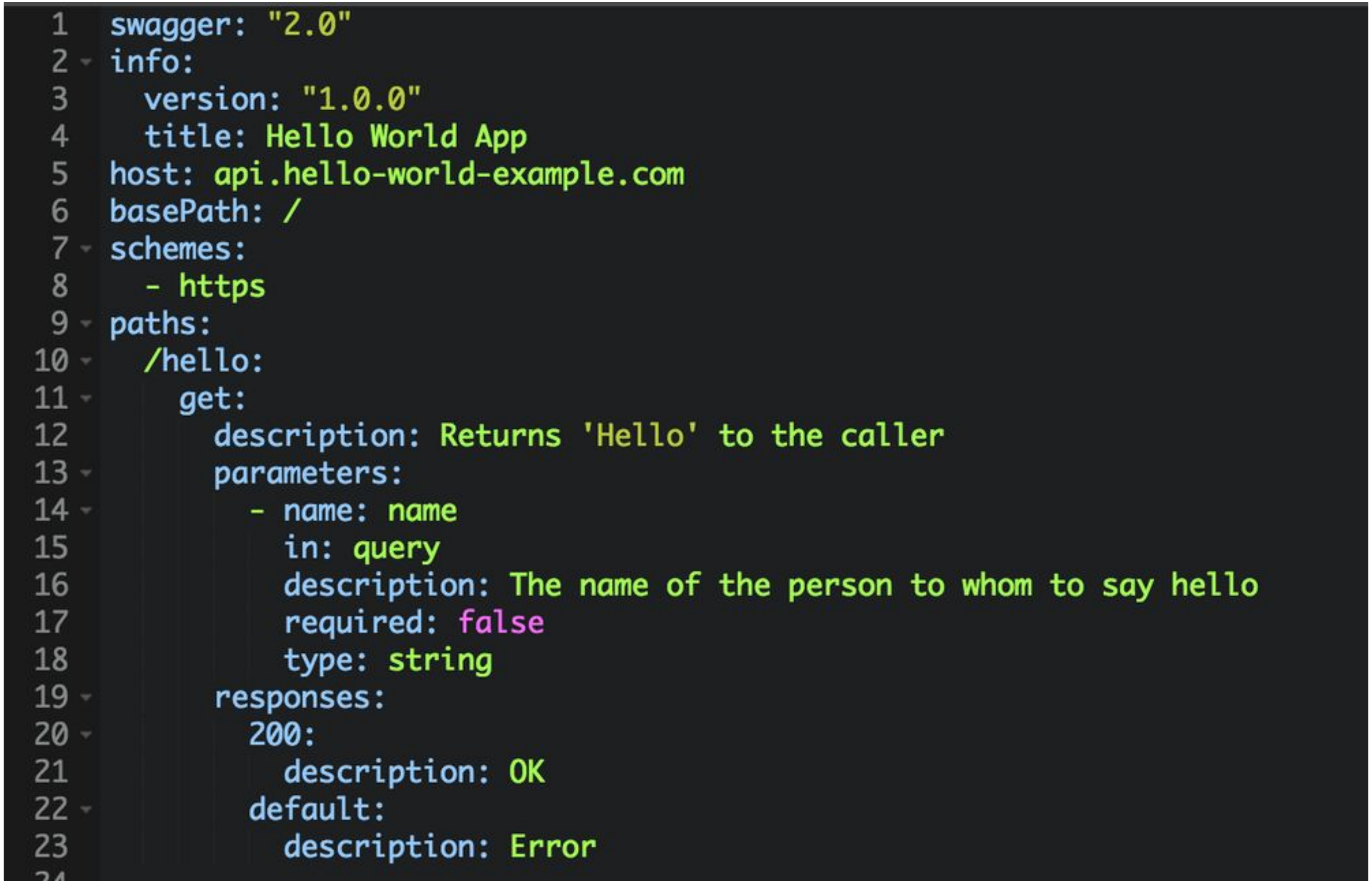
- Microservice typically must communicate with other microservices to provide its functionality
- Principle of least knowledge
  - Each unit should have only limited knowledge about other units: only units "closely" related to the current unit
  - Each unit should only talk to its friends; don't talk to strangers

Single responsibility principle

- “*micro*” doesn’t necessarily mean small
- Micro == scope of the service responsibility
- SRP - a class (microservice) should have only one reason to change
- Unix philosophy - Make each program (microservice) do one thing well

API

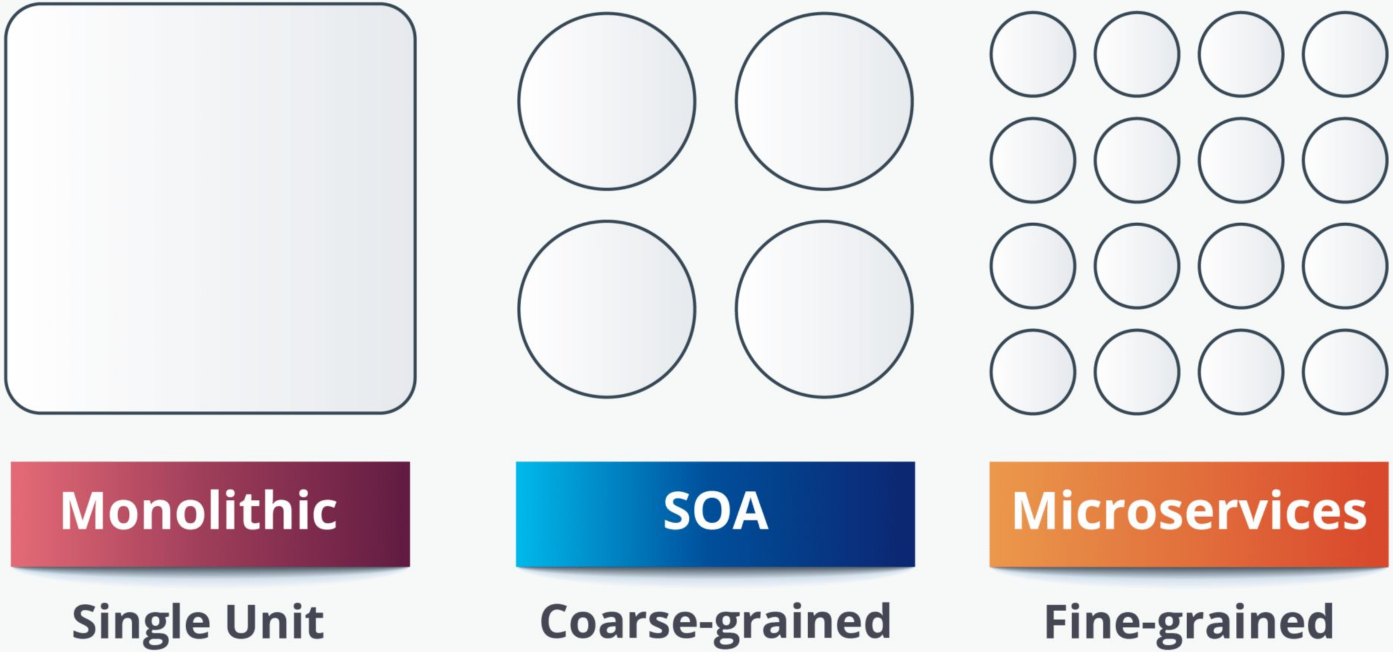
- Application programming interface
- Technology-agnostic
- Remote procedure calls
  - HTTP & REST
  - Apache Kafka
  - gRPC
  - AMQP, MQTT, JMS
  - ....



```yaml
swagger: "2.0"
info:
  version: "1.0.0"
  title: Hello World App
host: api.hello-world-example.com
basePath: /
schemes:
  - https
paths:
  /hello:
    get:
      description: Returns 'Hello' to the caller
      parameters:
        - name: name
          in: query
          description: The name of the person to whom to say hello
          required: false
          type: string
      responses:
        200:
          description: OK
        default:
          description: Error
```

Microservices vs SOA

Monolithic vs. SOA vs. Microservices



Monolithic

Single Unit

SOA

Coarse-grained

Microservices

Fine-grained

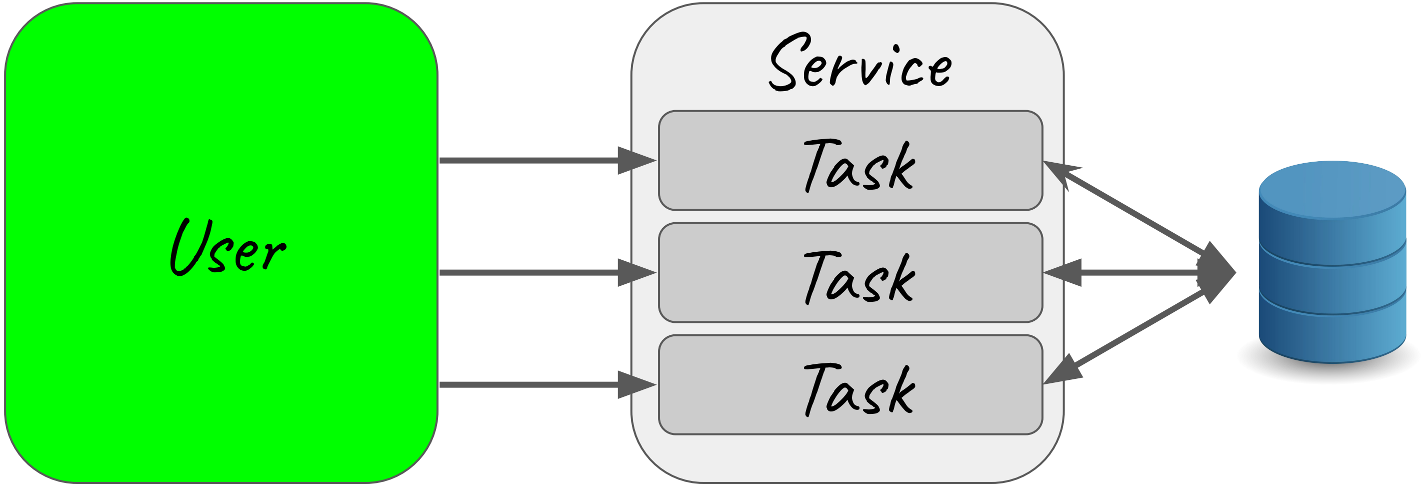
Microservices vs SOA

- Both are about services…

- but the service characteristics differ

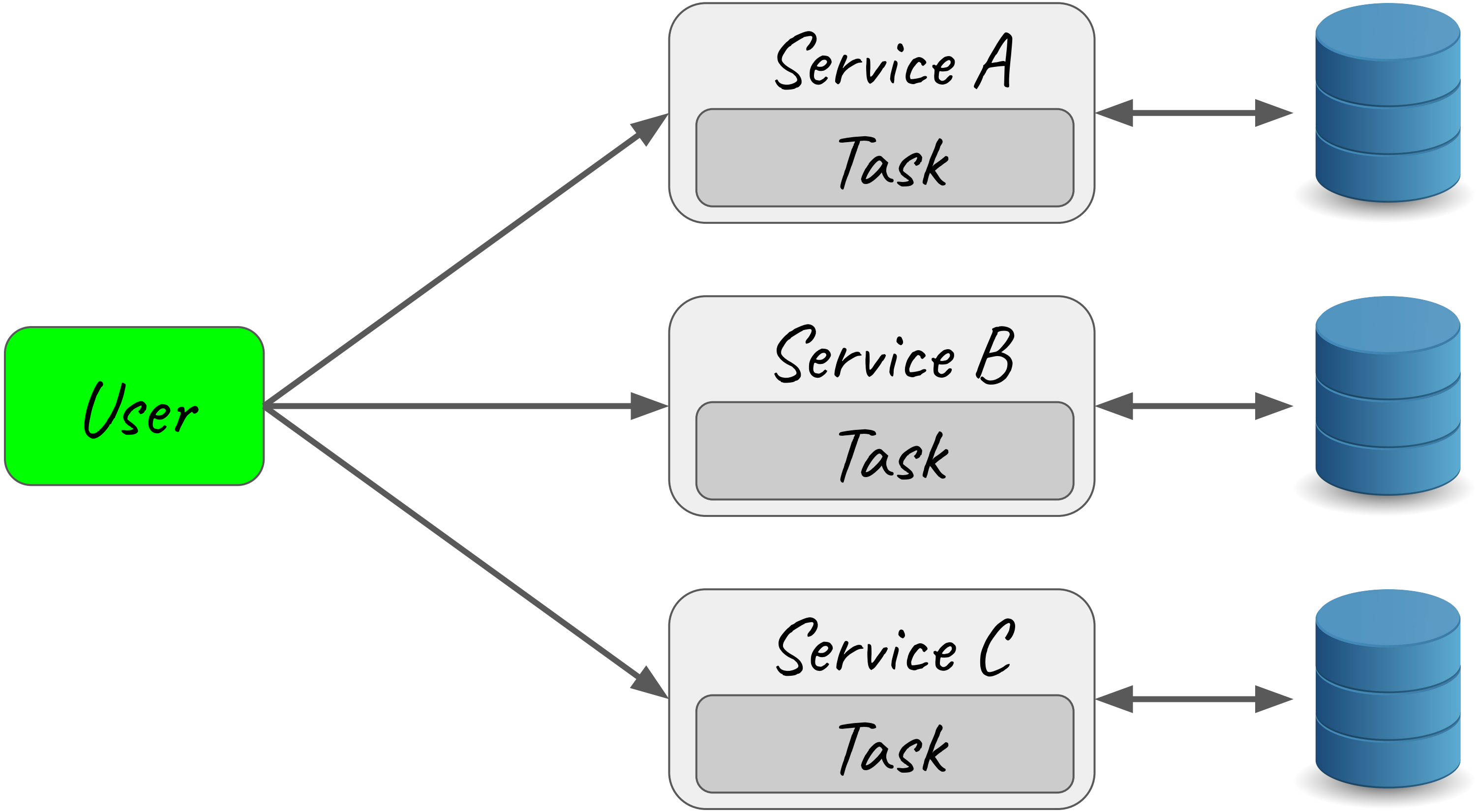
SOA

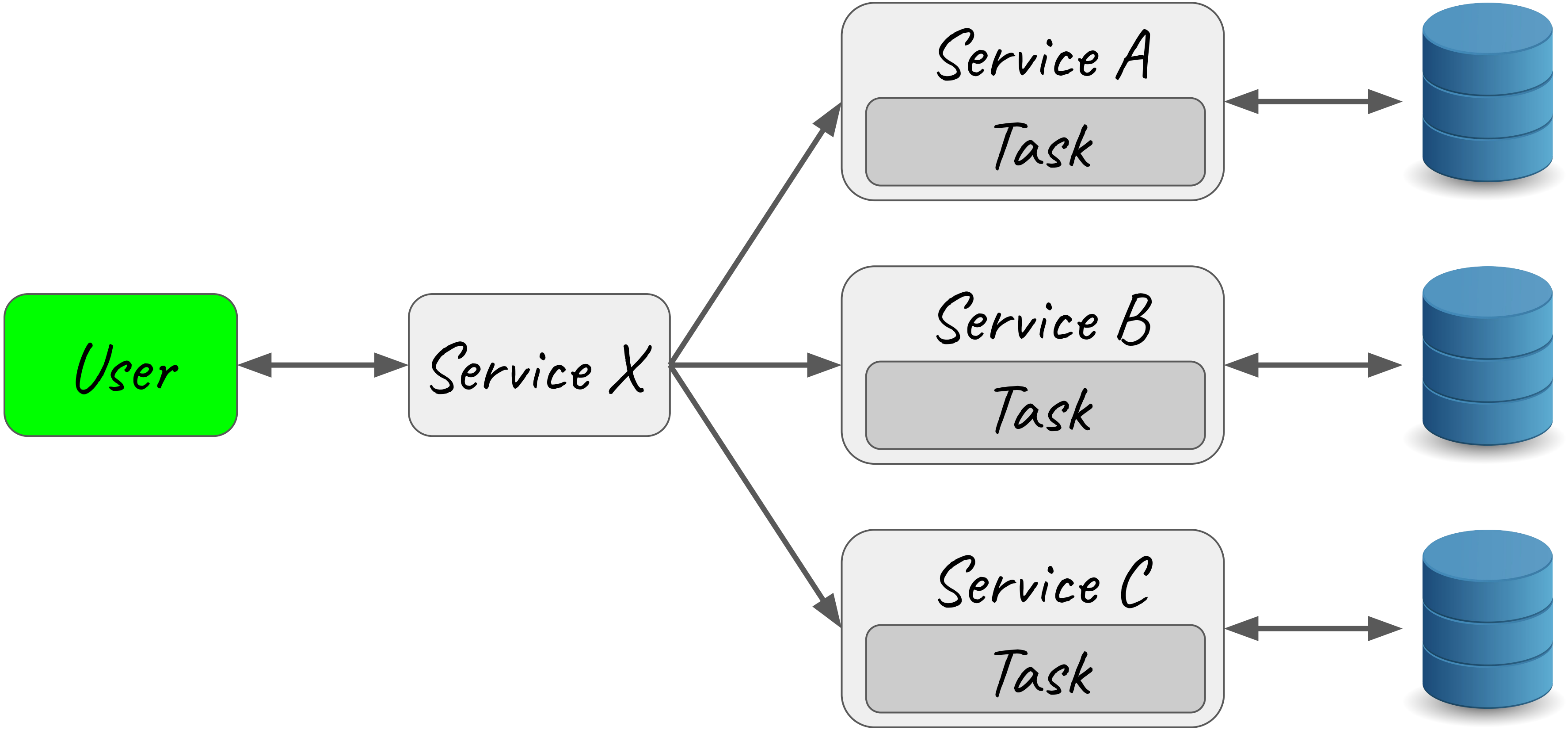
- Smaller number of services
- Each service is responsible for a particular group of functions/tasks that together provide a system functionality
- User requests are typically processed by one or in a small number of services
    - User can interact with the same service in several requests/steps

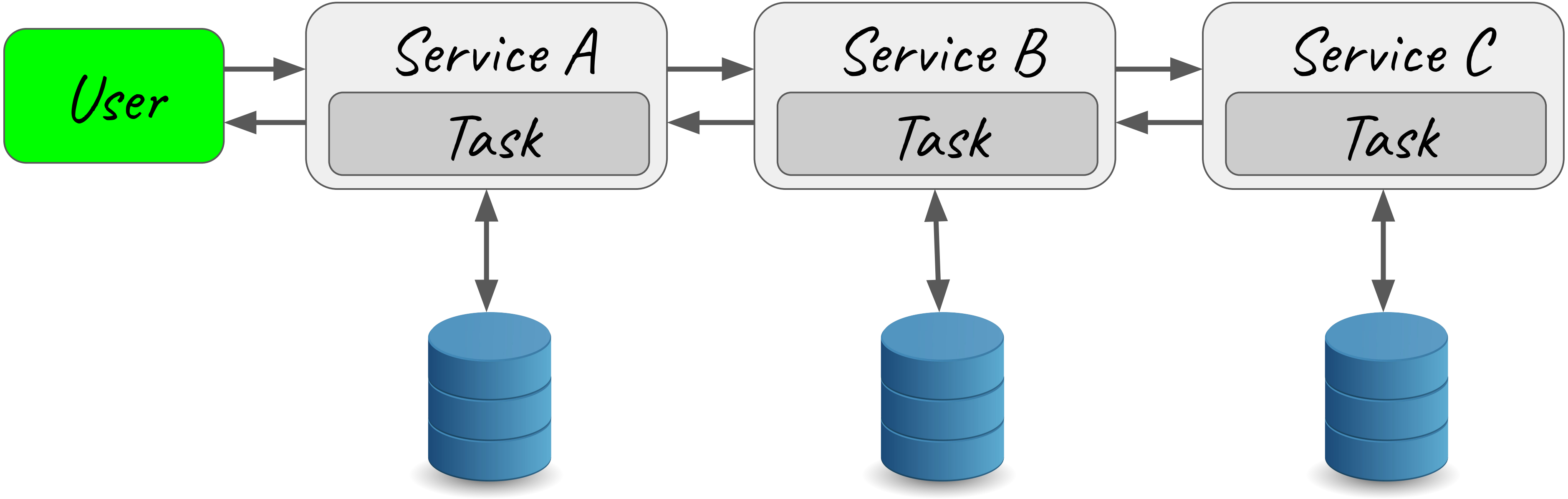

User
Service
Task
Task
Task

Microservices

- Higher number of services
- Each service is responsible for one particular function/task
- Users typically interacts with several services (not necessarily directly)
- The system provides its functions as an collaboration/interaction between microservices

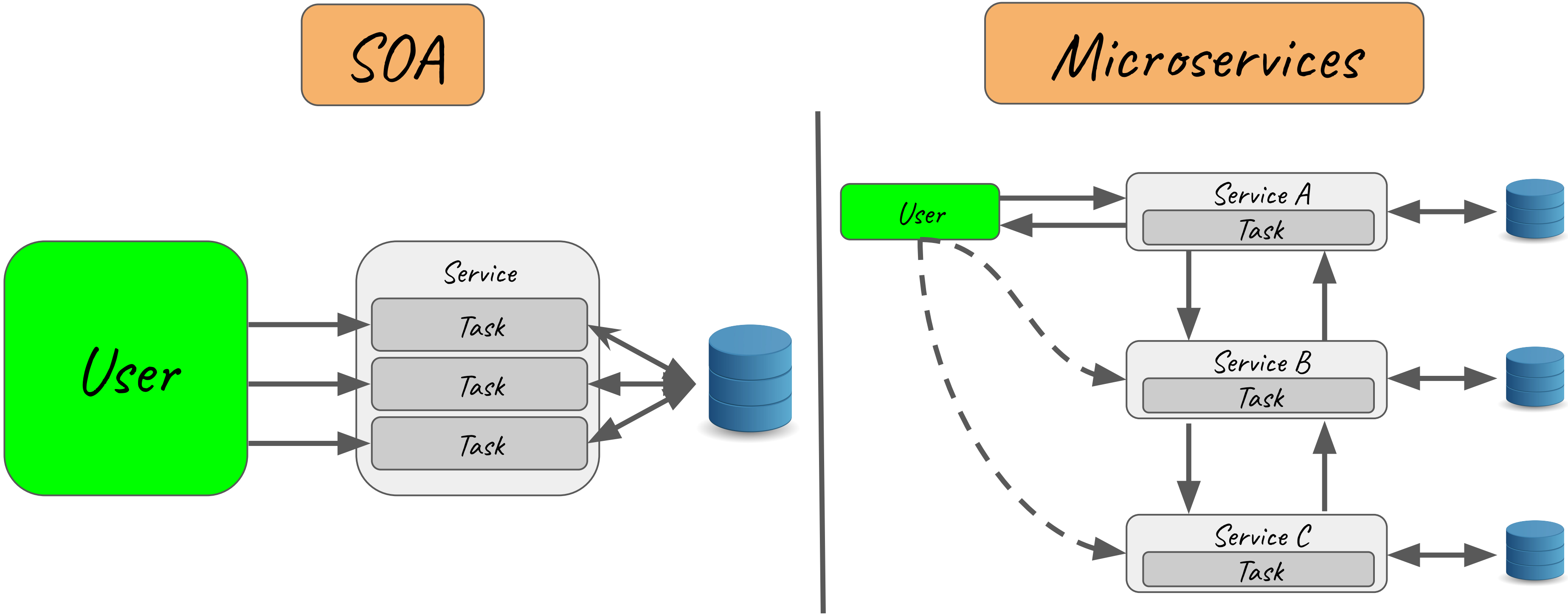

User
Service A
Task
Service B
Task
Service C
Task


User
Service X
Service A
Task
Service B
Task
Service C
Task

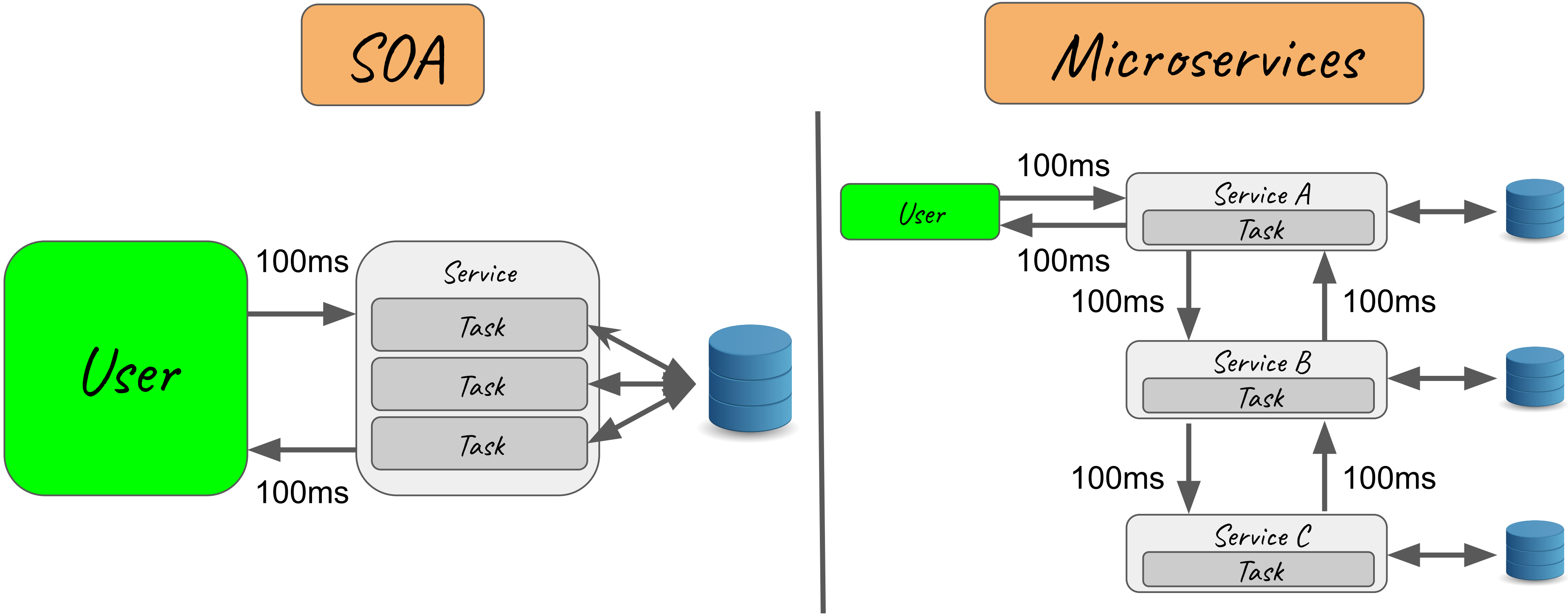

User
Service A
Task
Service B
Task
Service C
Task

Differences side-by-side

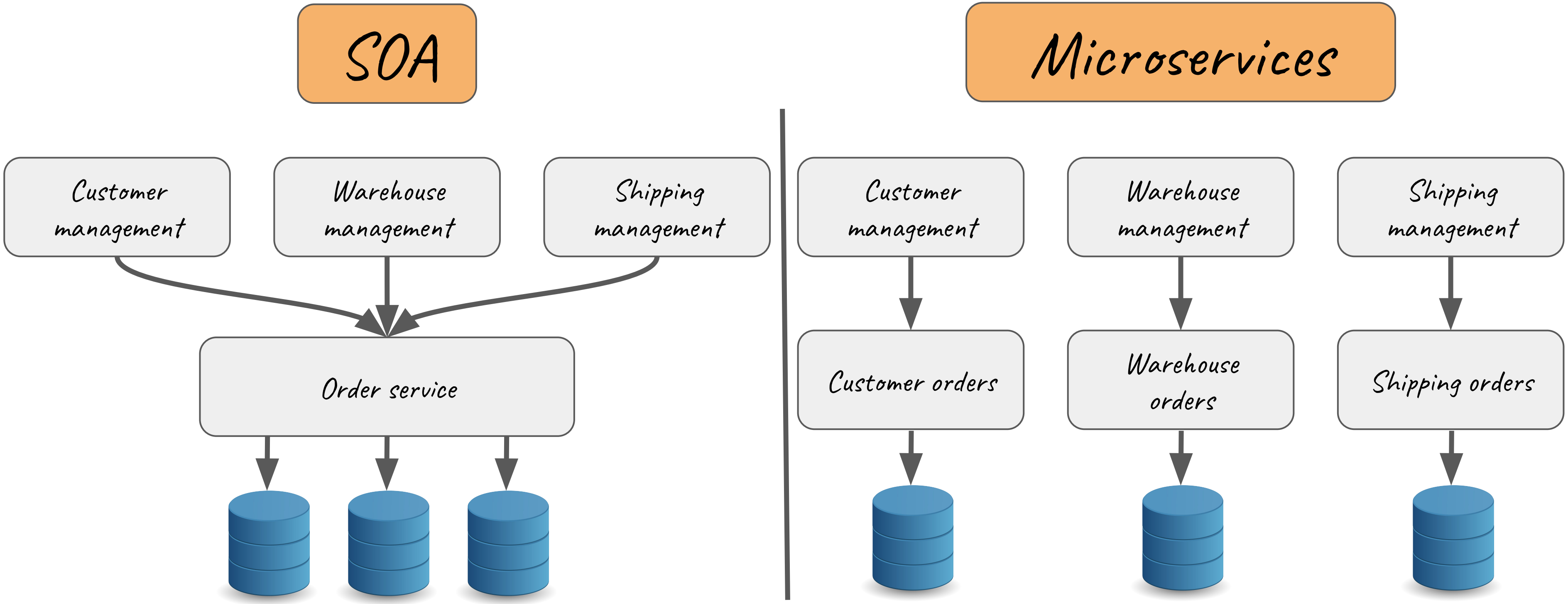
Service organization



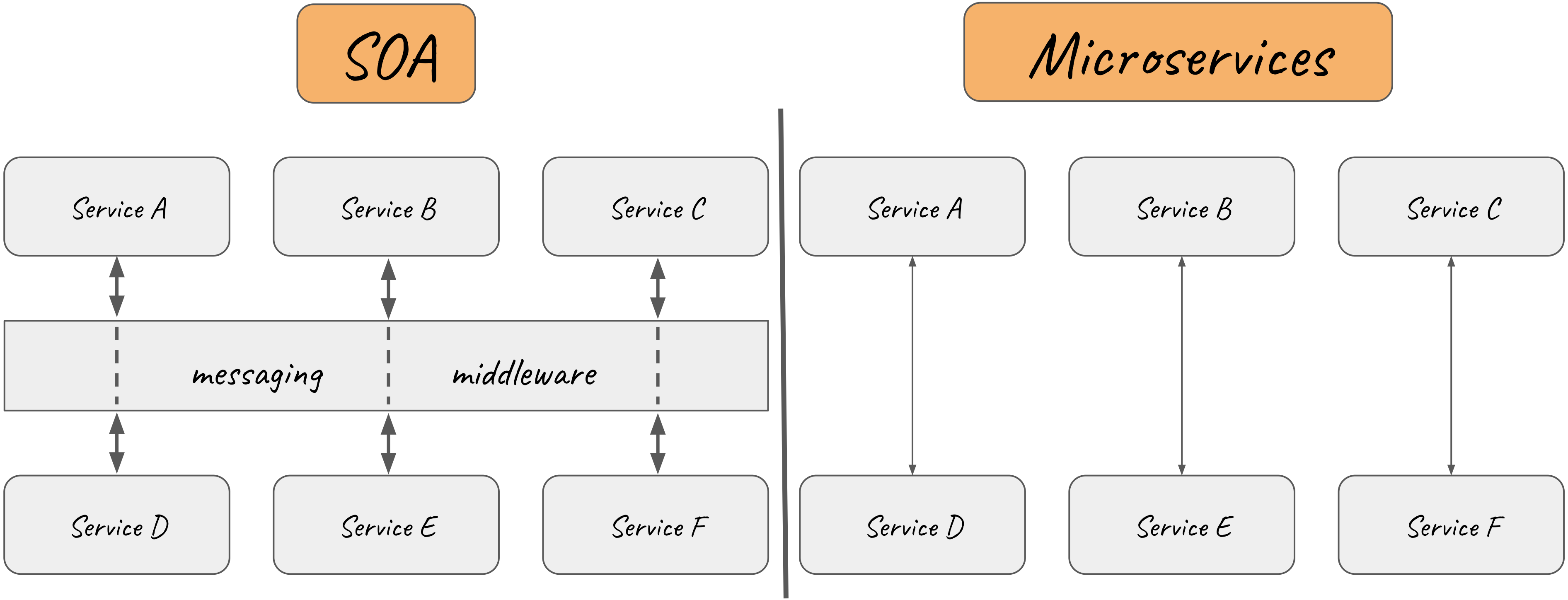
Task granularity



Component sharing



Message exchange



SOA

- Share **as much as possible**
- Importance on the **business functionality reuse**
- Common **governance** and **standards**

Microservices

- Share **as little as possible**
- Importance on **bounded contexts**
- Focus on **people, collaboration, freedom of options**

SOA

- **Enterprise service bus** (ESB)
- **Standardized messaging** protocols
- **Multi-threaded**

Microservices

- **Direct communication** (service mesh)
- **Lightweight protocols** (HTTP/REST, Kafka)
- **Single-threaded** (event loop)

SOA

- Maximize **service reusability**
- **Traditional technologies** (Relation DBs)

Microservices

- Focus on **service decoupling**
- **Modern technologies,** faster adoption (NoSQL)

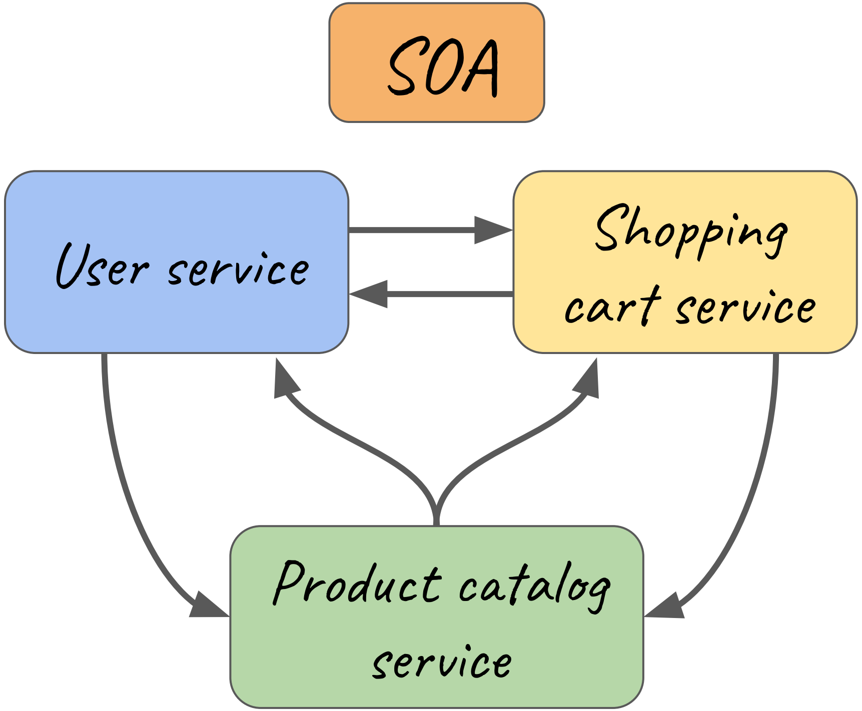
SOA

- Any change still resembles problems similar to monoliths
- Traditional development models

Microservices

- Any change is only local to a particular service (or add a new service)
- Strong focus on **DevOps** and **CI/CD**

Example

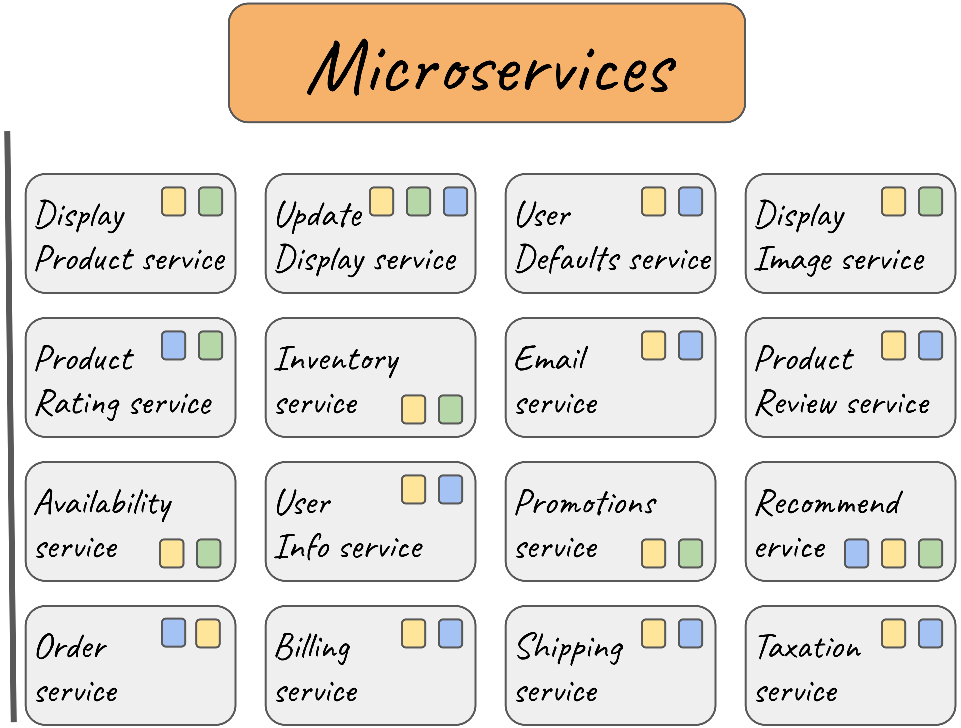


SOA

User service

Shopping cart service

Product catalog service



Microservices

Display Product service

Update Display service

User Defaults service

Display Image service

Product Rating service

Inventory service

Email service

Product Review service

Availability service

User Info service

Promotions service

Recommend ervice

Order service

Billing service

Shipping service

Taxation service

Technologies

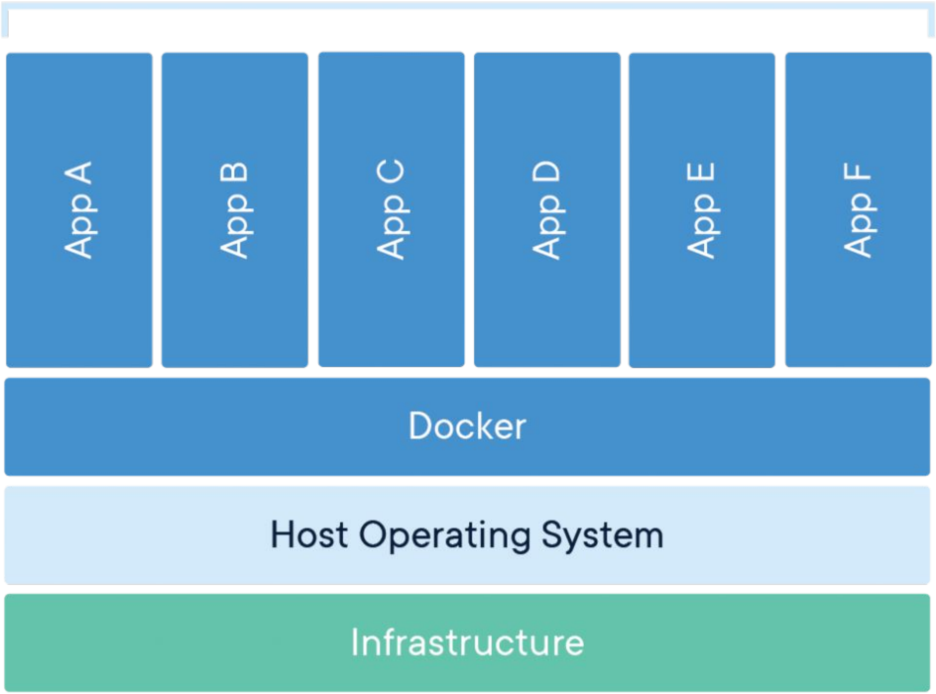
Docker (containers)

- **Container** - a standardized unit of software
- Packages code and its dependencies, runtime, system tools, system libraries, settings
- Users build **Docker images** - lightweight, standalone, executable package of software
- Images can be run anywhere where Docker is installed
- **Docker hub** (hub.docker.com)

Docker containers

- Container is runtime representation of the image
- Containers run on Docker Engine
- It doesn’t matter on which platform (Linux, Mac, Windows) you run
- Containers isolate software from its environment
- Uniform behavior everywhere

Containerized Applications



Docker Engine

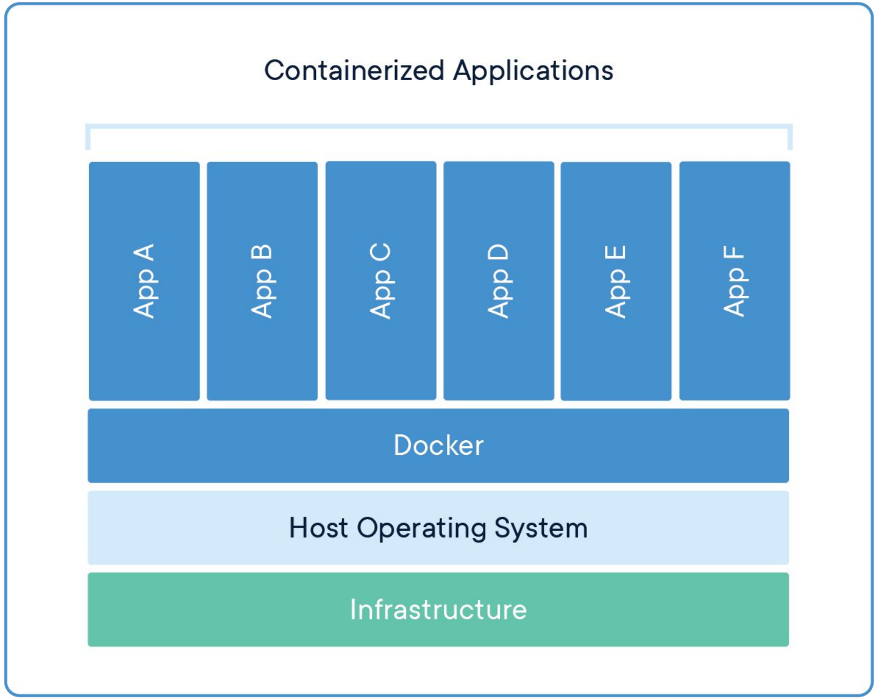
- **Standard** - Docker created the industry standard for containers, so they could be portable anywhere

- **Lightweight** - Containers share the machine's OS system kernel and therefore do not require an OS per application

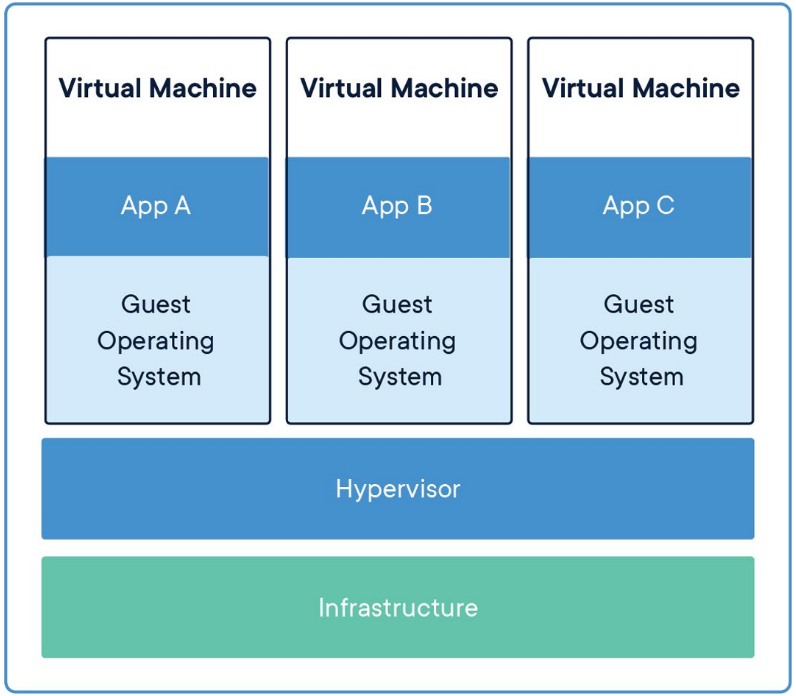
- **Secure**: Applications are safer in containers and Docker provides the strongest default isolation capabilities in the industry

Containers vs Virtual Machines

Containers and virtual machines have similar resource isolation and allocation benefits, but function differently because **containers virtualize the operating system instead of hardware**. Containers are more portable and efficient.



Containerized Applications
App A
App B
App C
App D
App E
App F
Docker
Host Operating System
Infrastructure

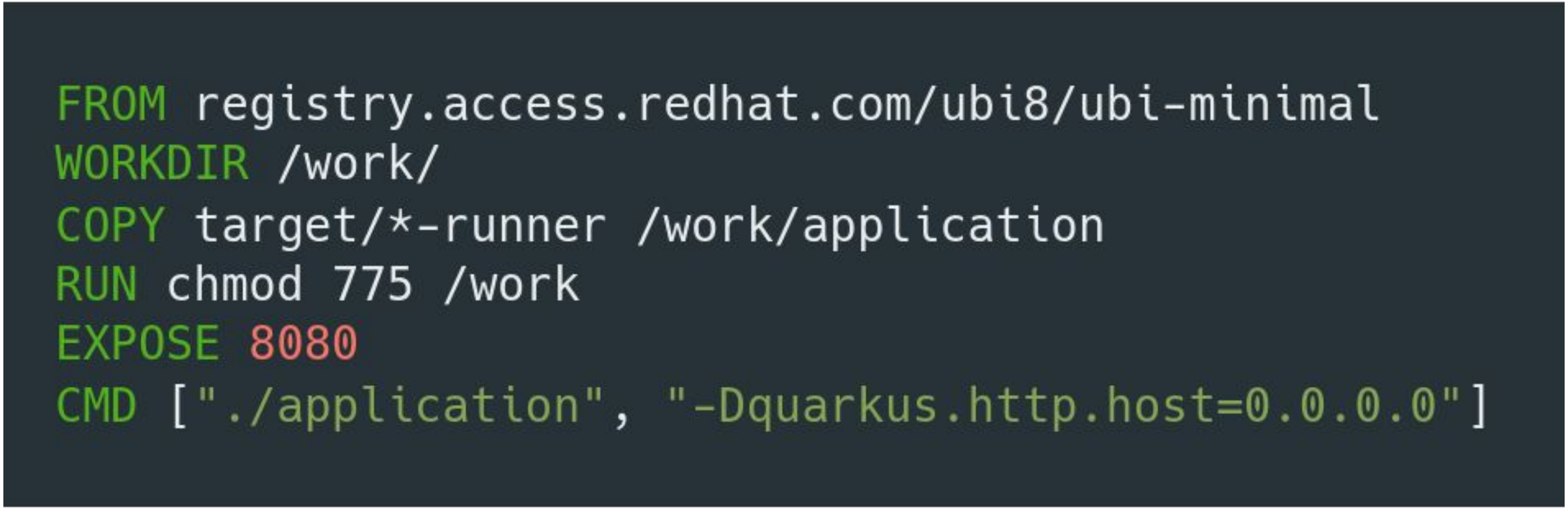


Virtual Machine
Virtual Machine
Virtual Machine
App A
App B
App C
Guest Operating System
Guest Operating System
Guest Operating System
Hypervisor
Infrastructure

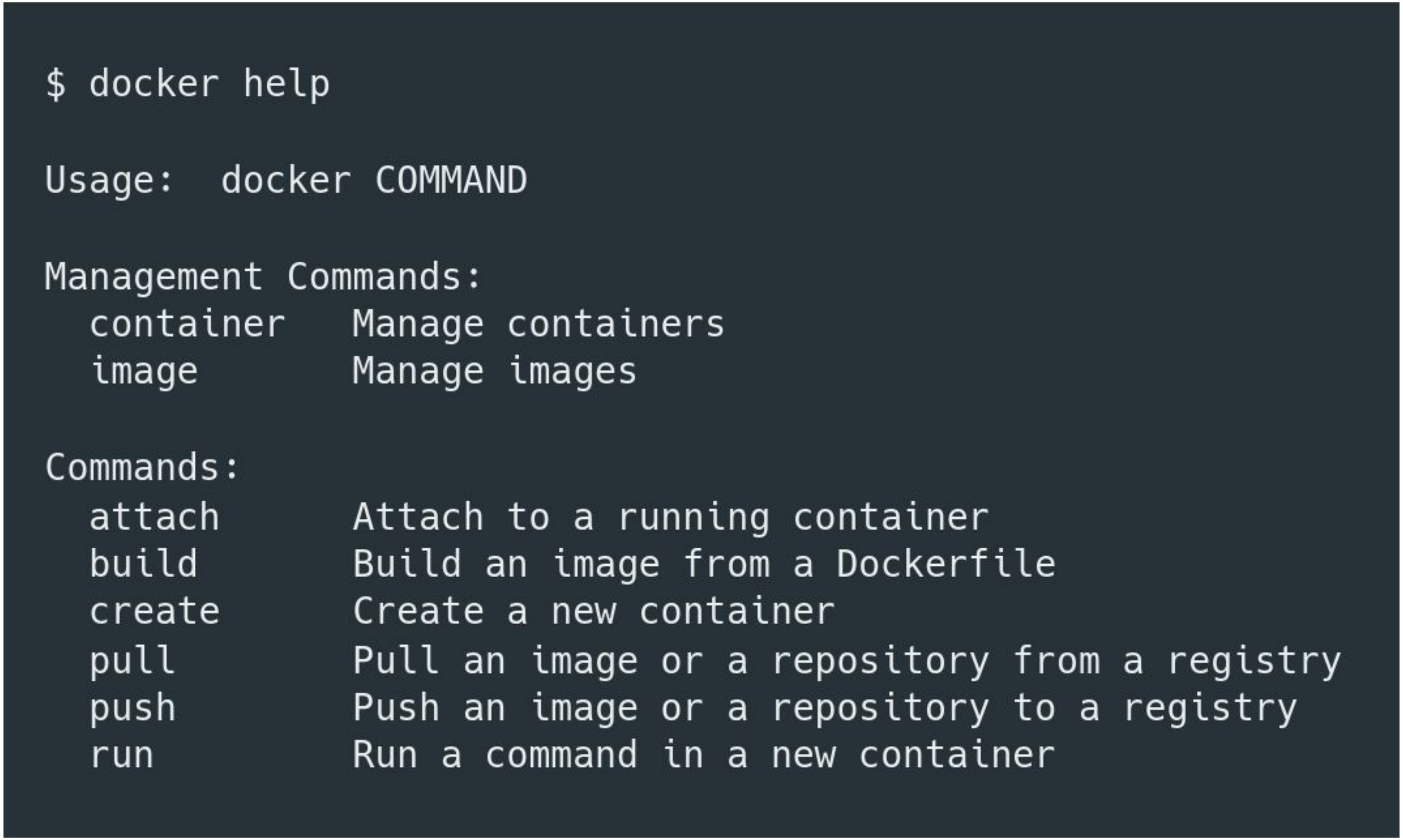
Docker - standardization

- Docker launched in 2013 - revolution in application development
- In June 2015, Docker donated the container image specification and runtime code now known as runc, to the **Open Container Initiative (OCI)**
- Other alternatives - Podman, Buildah

Dockerfile



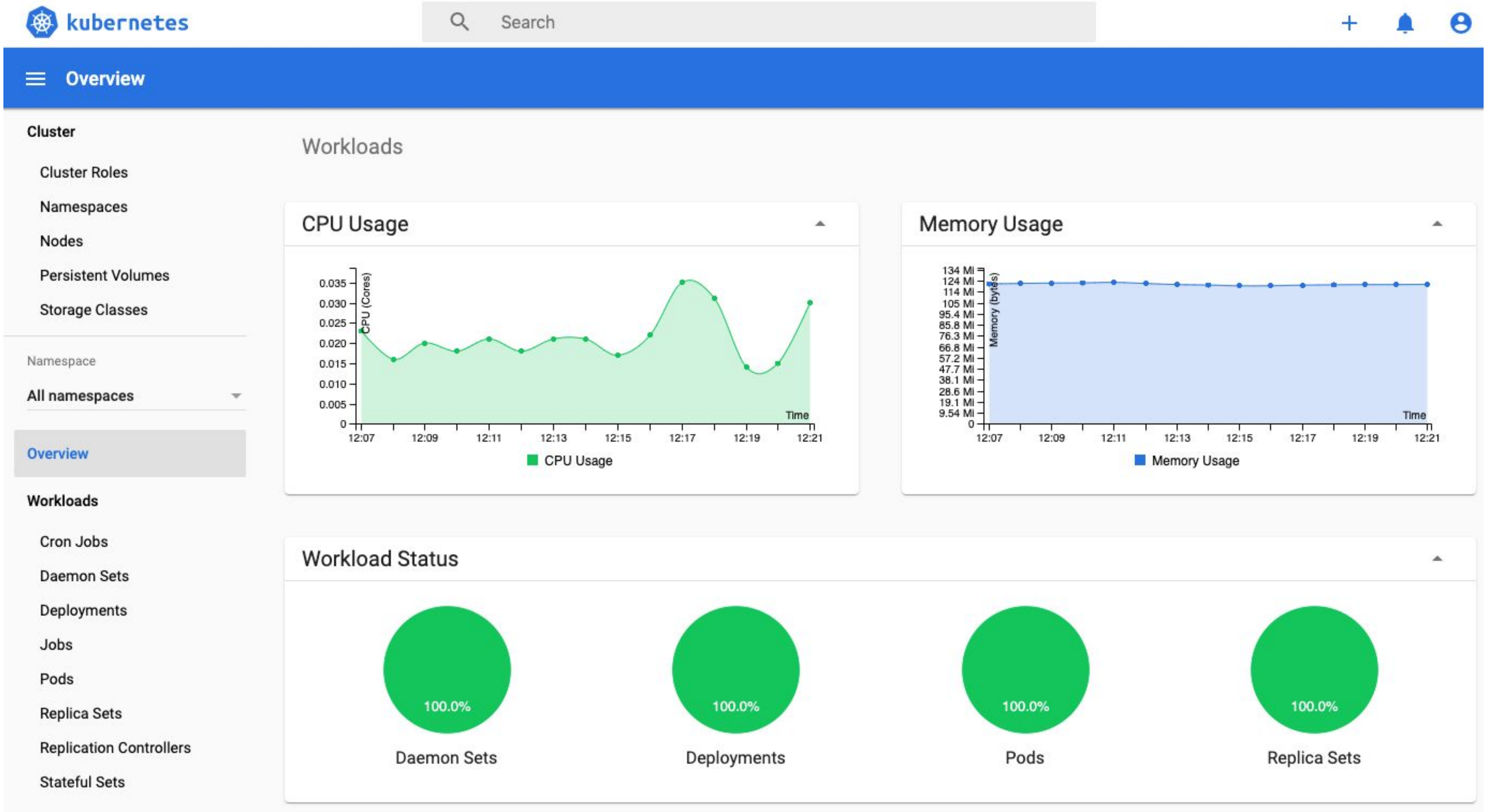
```
FROM registry.access.redhat.com/ubi8/ubi-minimal
WORKDIR /work/
COPY target/*-runner /work/application
RUN chmod 775 /work
EXPOSE 8080
CMD ["./application", "-Dquarkus.http.host=0.0.0.0"]
```



```
$ docker help

Usage:  docker COMMAND

Management Commands:
  container   Manage containers
  image       Manage images

Commands:
  attach      Attach to a running container
  build       Build an image from a Dockerfile
  create      Create a new container
  pull        Pull an image or a repository from a registry
  push        Push an image or a repository to a registry
  run         Run a command in a new container
```

Kubernetes

- Container orchestration
- an open-source system for automating deployment, scaling, and management of containerized applications
- Groups containers to logical units
- Easy administration
- De facto standard for cloud deployments

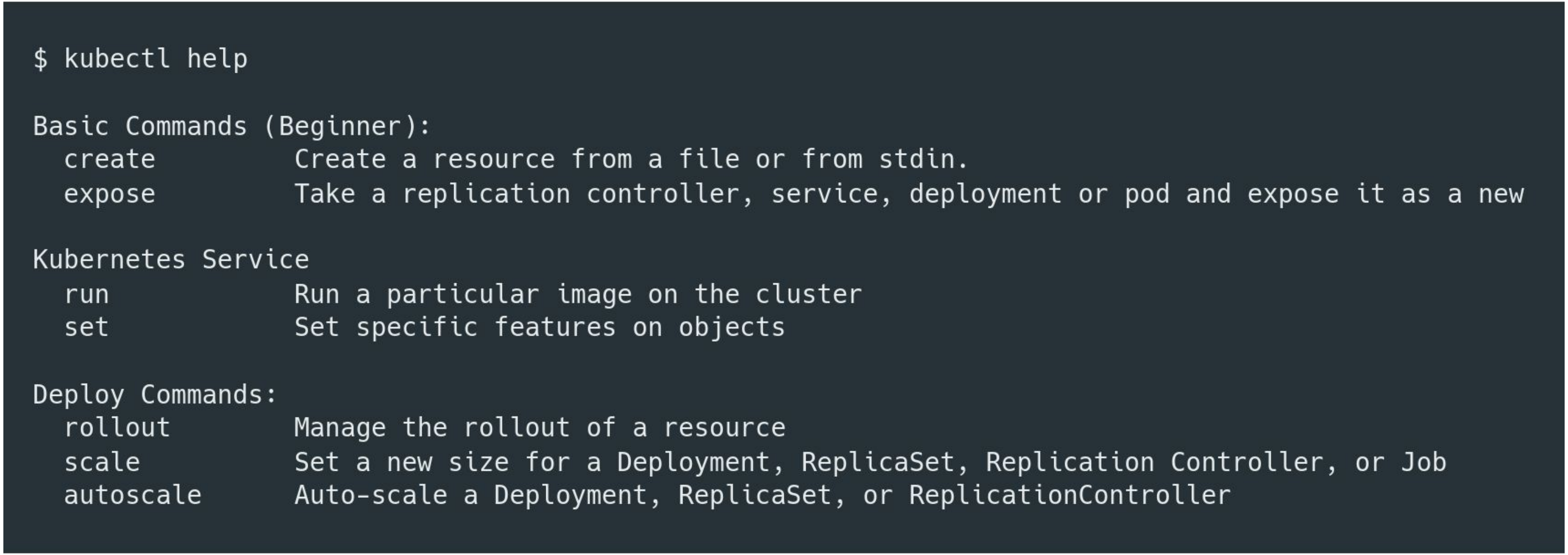

kubernetes
Search
Overview
Cluster
Cluster Roles
Namespaces
Nodes
Persistent Volumes
Storage Classes
Namespace
All namespaces
Overview
Workloads
Cron Jobs
Daemon Sets
Deployments
Jobs
Pods
Replica Sets
Replication Controllers
Stateful Sets
Workloads
CPU Usage
0.035
0.030
0.025
0.020
0.015
0.010
0.005
0
CPU (Cores)
Time
12:07
12:09
12:11
12:13
12:15
12:17
12:19
12:21
🟩 CPU Usage
Memory Usage
134 Mi
124 Mi
114 Mi
105 Mi
95.4 Mi
85.8 Mi
76.3 Mi
66.8 Mi
57.2 Mi
47.7 Mi
38.1 Mi
28.6 Mi
19.1 Mi
9.54 Mi
0
Memory (bytes)
Time
12:07
12:09
12:11
12:13
12:15
12:17
12:19
12:21
🟦 Memory Usage
Workload Status
100.0%
Daemon Sets
100.0%
Deployments
100.0%
Pods
100.0%
Replica Sets

Kubernetes objects

- **Pod** – basic executions unit
  - Process running in the cluster
  - One or multiple containers
  - Replaceable unit, can be restarted anytime (health checks)
- **Service** – exposure of application (pods) as a network service
  - Abstraction of the access to pods

Kubernetes objects

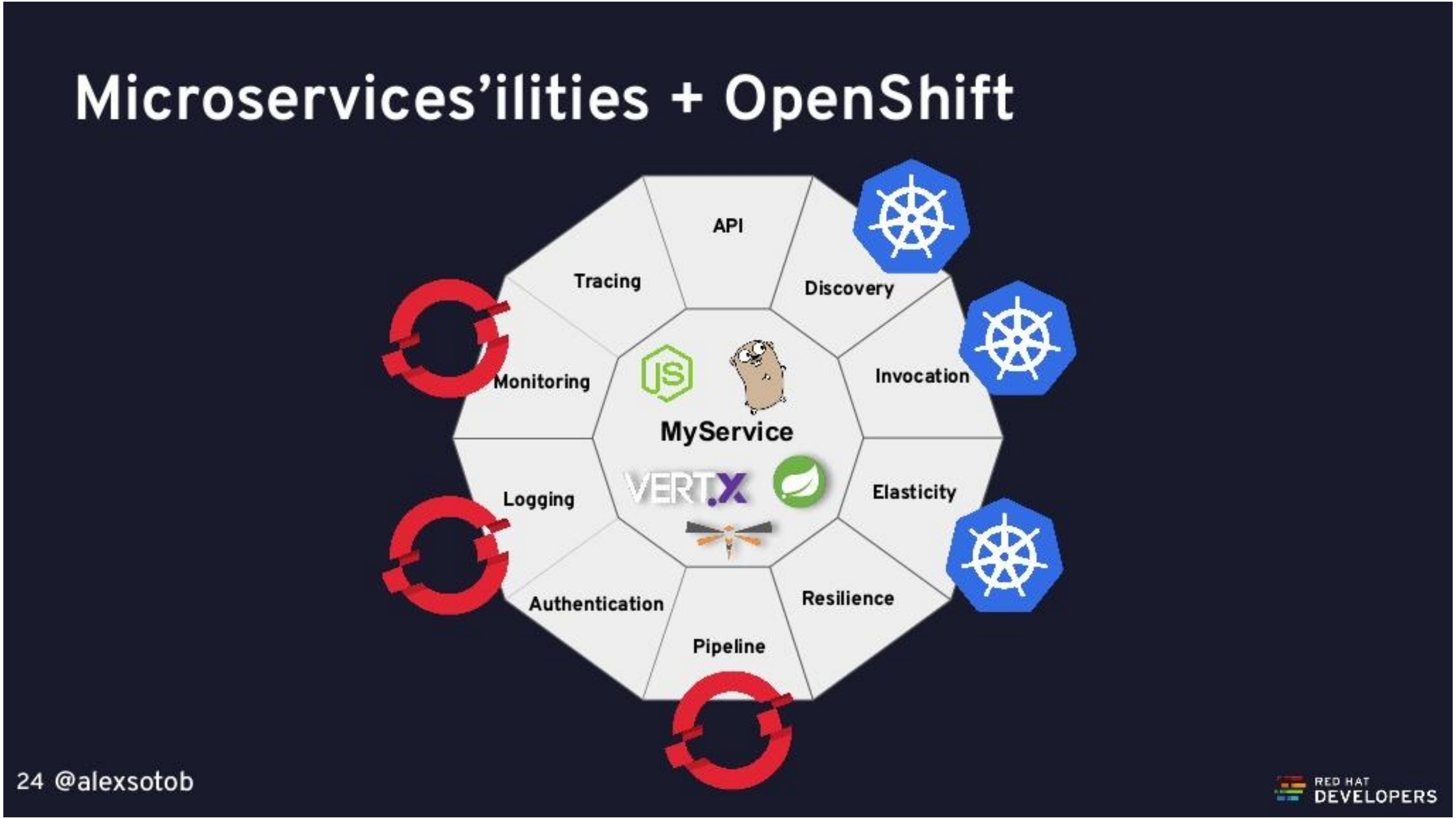
- **Volume** - storage shared between containers in the pod
- **Deployment** - declarative updates for pods
  - User describes the desired state
  - Deployment controller (dc) changes the actual state to the desired state at controlled rate
  - New state of the pods, rollbacks, scaling,...



```
$ kubectl help

Basic Commands (Beginner):
  create         Create a resource from a file or from stdin.
  expose         Take a replication controller, service, deployment or pod and expose it as a new

Kubernetes Service
  run            Run a particular image on the cluster
  set            Set specific features on objects

Deploy Commands:
  rollout        Manage the rollout of a resource
  scale          Set a new size for a Deployment, ReplicaSet, Replication Controller, or Job
  autoscale      Auto-scale a Deployment, ReplicaSet, or ReplicationController
```

OpenShift

- Fork of Kubernetes developed and maintained at Red Hat
- Commercial product with support
- Automated installation, upgrades, and lifecycle management throughout the container stack



Microservices'ilities + OpenShift

API

Discovery

Invocation

Elasticity

Resilience

Pipeline

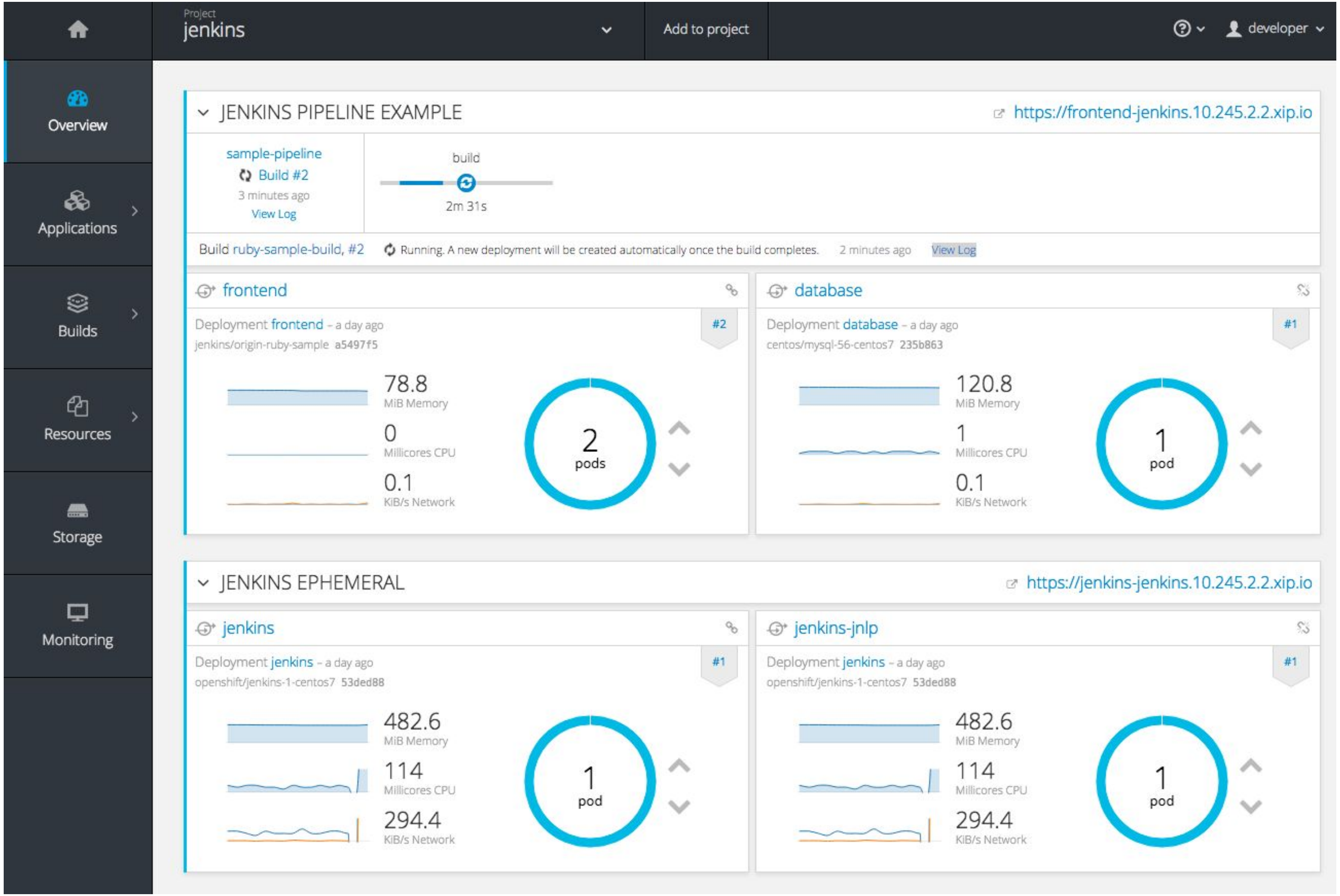
Authentication

Logging

Monitoring

Tracing

MyService



Project
jenkins

JENKINS PIPELINE EXAMPLE

https://frontend-jenkins.10.245.2.2.xip.io

sample-pipeline
Build #2
3 minutes ago
View Log

build
2m 31s

Build ruby-sample-build, #2 Running. A new deployment will be created automatically once the build completes. 2 minutes ago View Log

frontend

Deployment frontend – a day ago #2
jenkins/origin-ruby-sample a5497f5

78.8
MiB Memory
0
Millicores CPU
0.1
KiB/s Network

2
pods

database

Deployment database – a day ago #1
centos/mysql-56-centos7 235b863

120.8
MiB Memory
1
Millicores CPU
0.1
KiB/s Network

1
pod

JENKINS EPHEMERAL

https://jenkins-jenkins.10.245.2.2.xip.io

jenkins

Deployment jenkins – a day ago #1
openshift/jenkins-1-centos7 53ded88

482.6
MiB Memory
114
Millicores CPU
294.4
KiB/s Network

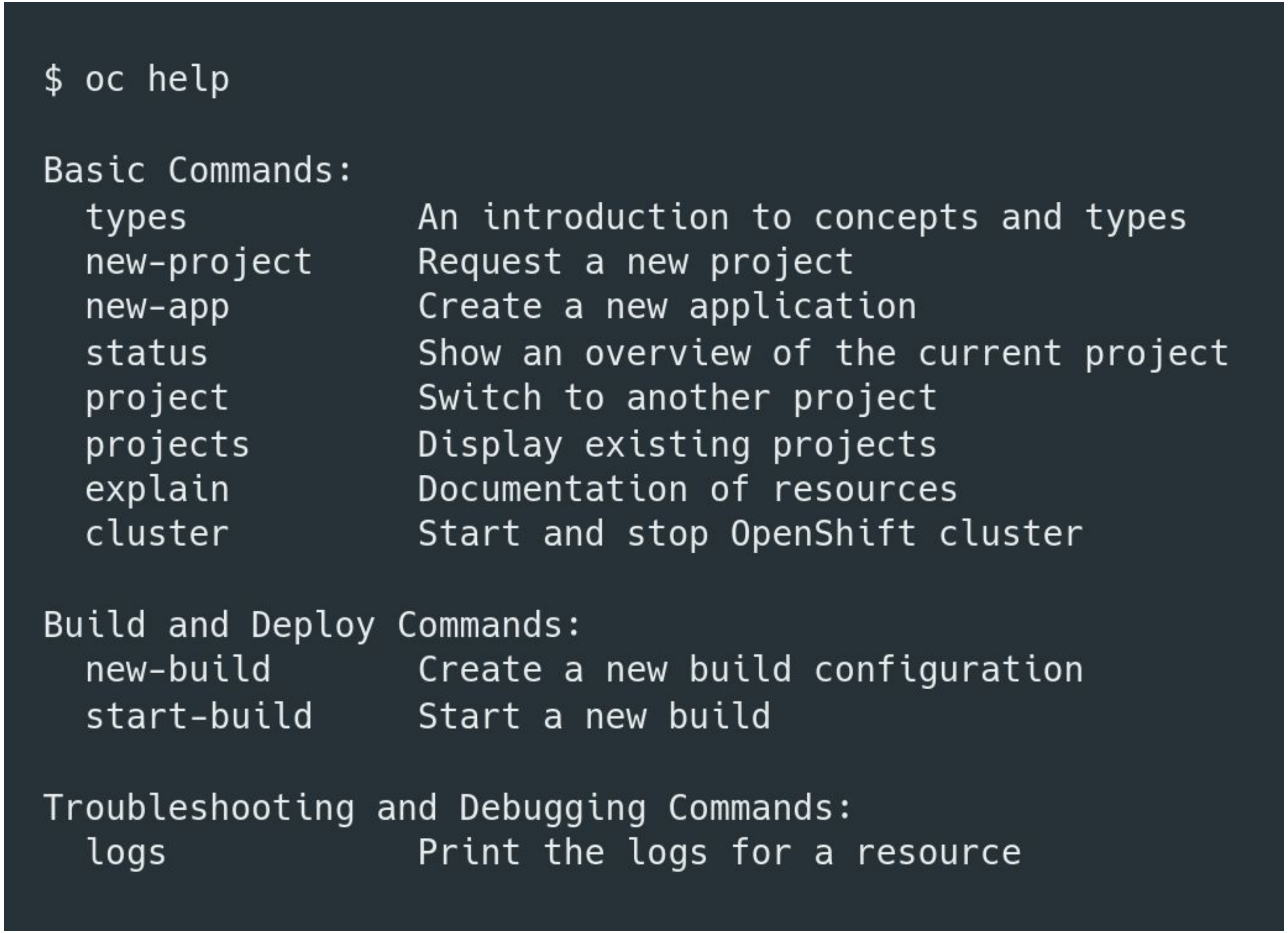
1
pod

jenkins-jnlp

Deployment jenkins – a day ago #1
openshift/jenkins-1-centos7 53ded88

482.6
MiB Memory
114
Millicores CPU
294.4
KiB/s Network

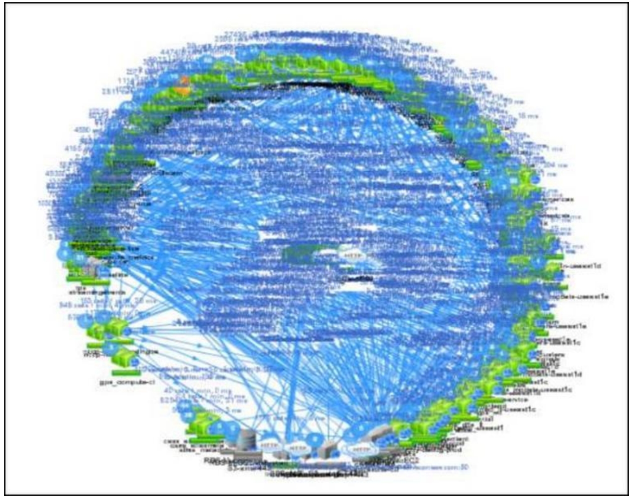
1
pod



```
$ oc help

Basic Commands:
  types             An introduction to concepts and types
  new-project       Request a new project
  new-app           Create a new application
  status            Show an overview of the current project
  project           Switch to another project
  projects          Display existing projects
  explain           Documentation of resources
  cluster           Start and stop OpenShift cluster

Build and Deploy Commands:
  new-build         Create a new build configuration
  start-build       Start a new build

Troubleshooting and Debugging Commands:
  logs              Print the logs for a resource
```

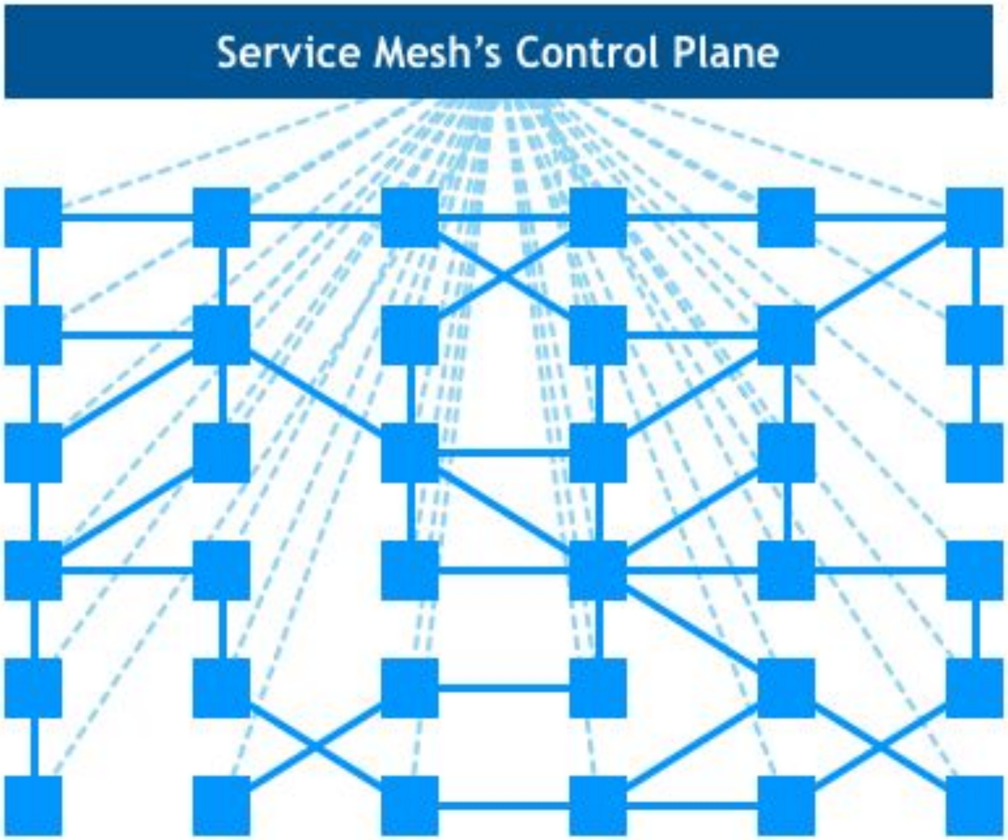
Istio – service mesh

- **Service mesh** – the network of microservices that make up the application and the interactions between them



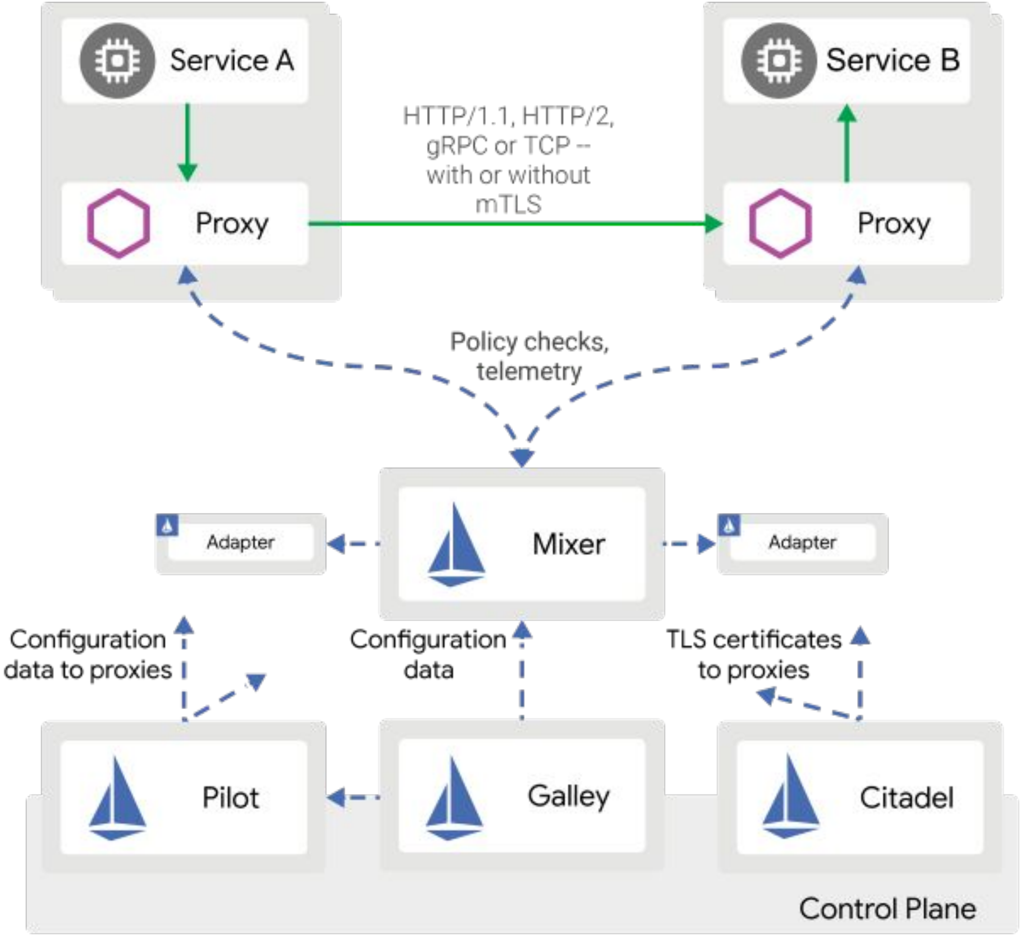
Istio - service mesh

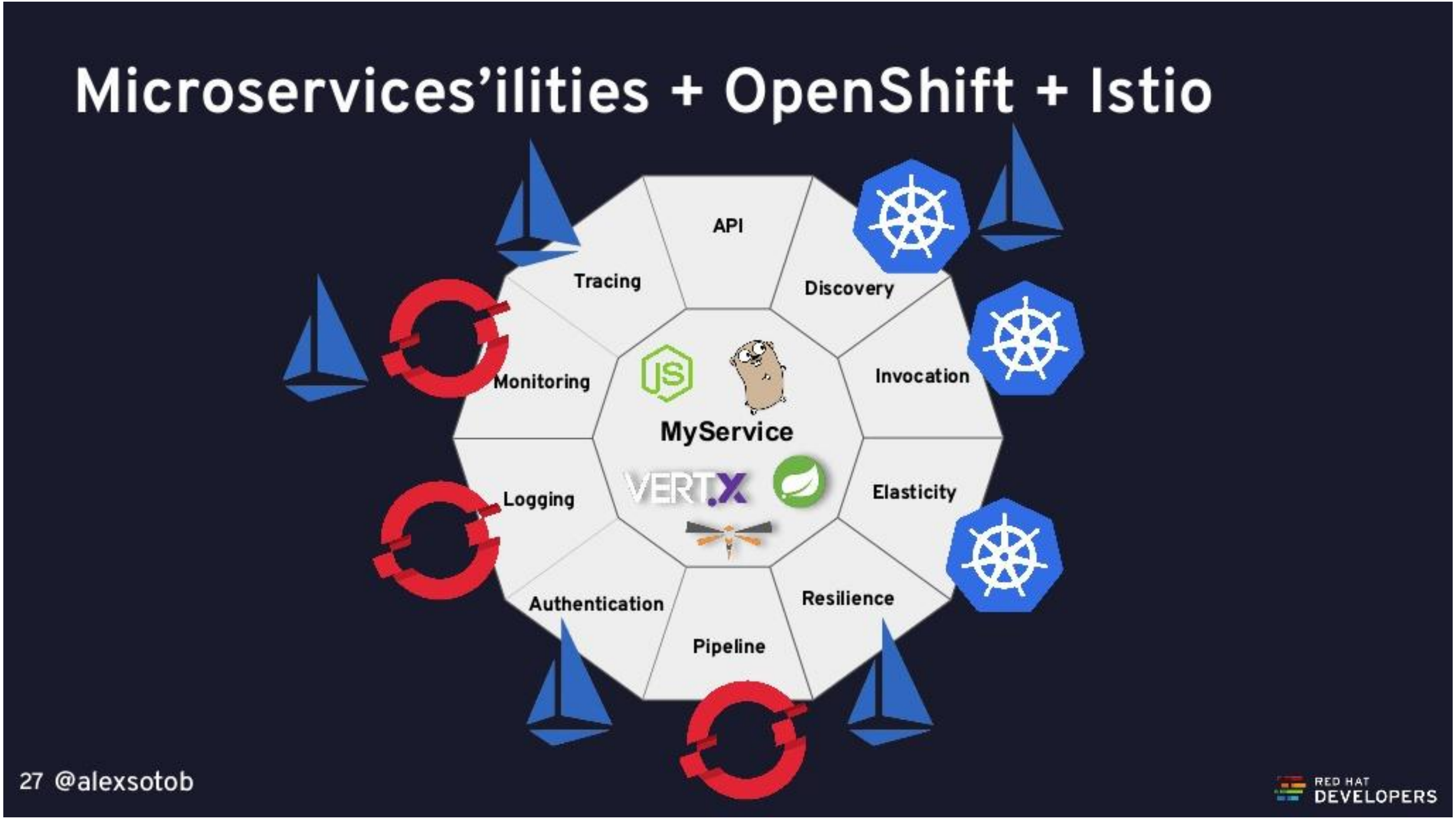
- As a service mesh grows in **size and complexity**, it can become **harder to understand and manage**
- requirements include **discovery**, **load balancing**, **failure recovery**, **metrics**, **and monitoring**
- operational requirements, like **A/B testing**, **canary rollouts**, **rate limiting**, **access control**, **and end-to-end authentication**


Service Mesh's Control Plane

Istio – Envoy

- Sidecar container
- Deployed in the same pod as the application container
- All network traffic goes through the Envoy proxy


Service A
Service B
HTTP/1.1, HTTP/2, gRPC or TCP -- with or without mTLS
Proxy
Proxy
Policy checks, telemetry
Adapter
Mixer
Adapter
Configuration data to proxies
Configuration data
TLS certificates to proxies
Pilot
Galley
Citadel
Control Plane



Microservices'ilities + OpenShift + Istio

API

Tracing

Discovery

Monitoring

Invocation

MyService

Logging

Elasticity

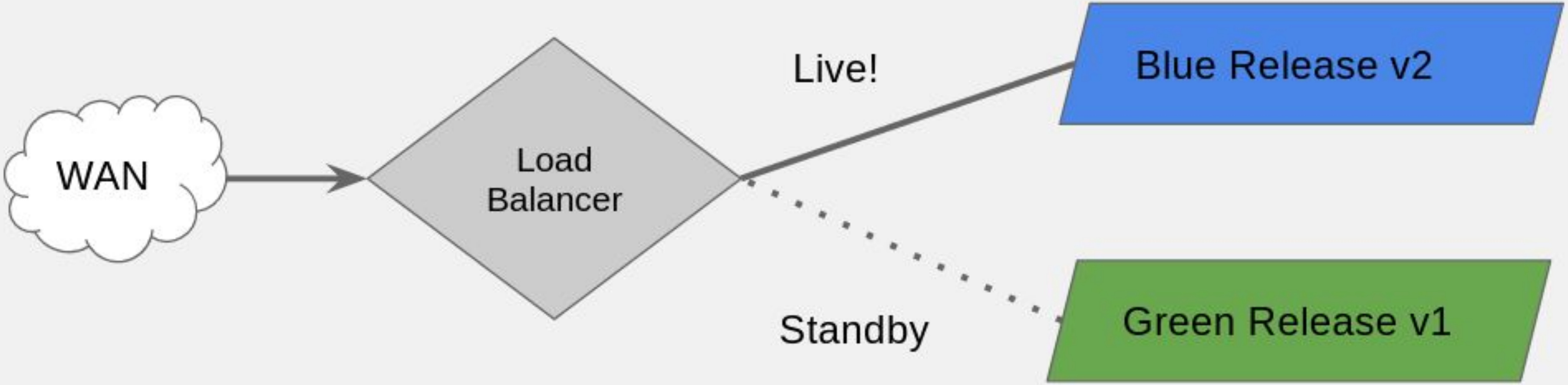
Authentication

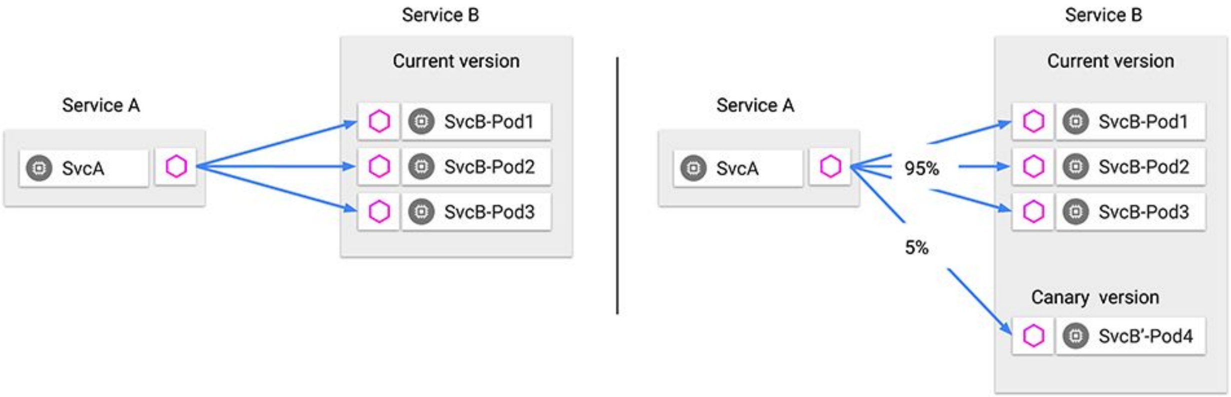
Resilience

Pipeline

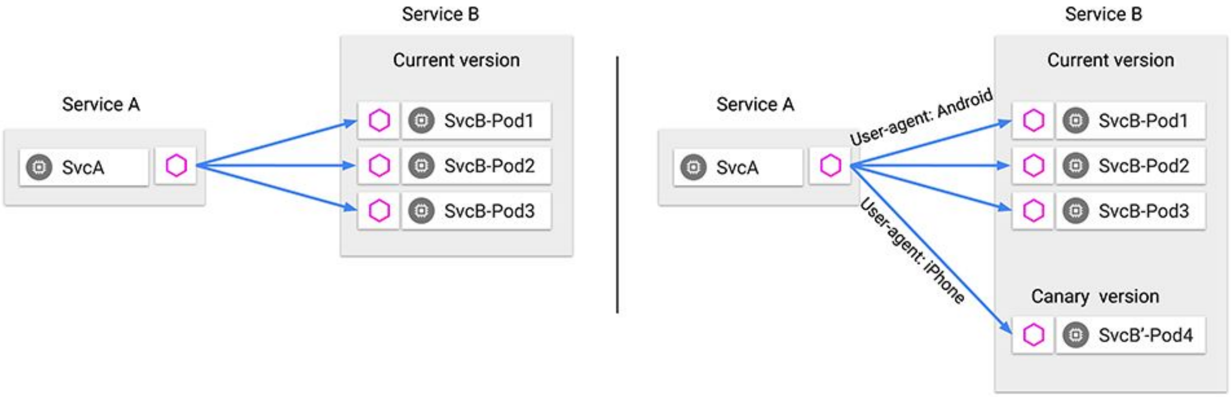
Blue / Green Deployment

Keep a hot standby ready in case a new release is flawed.

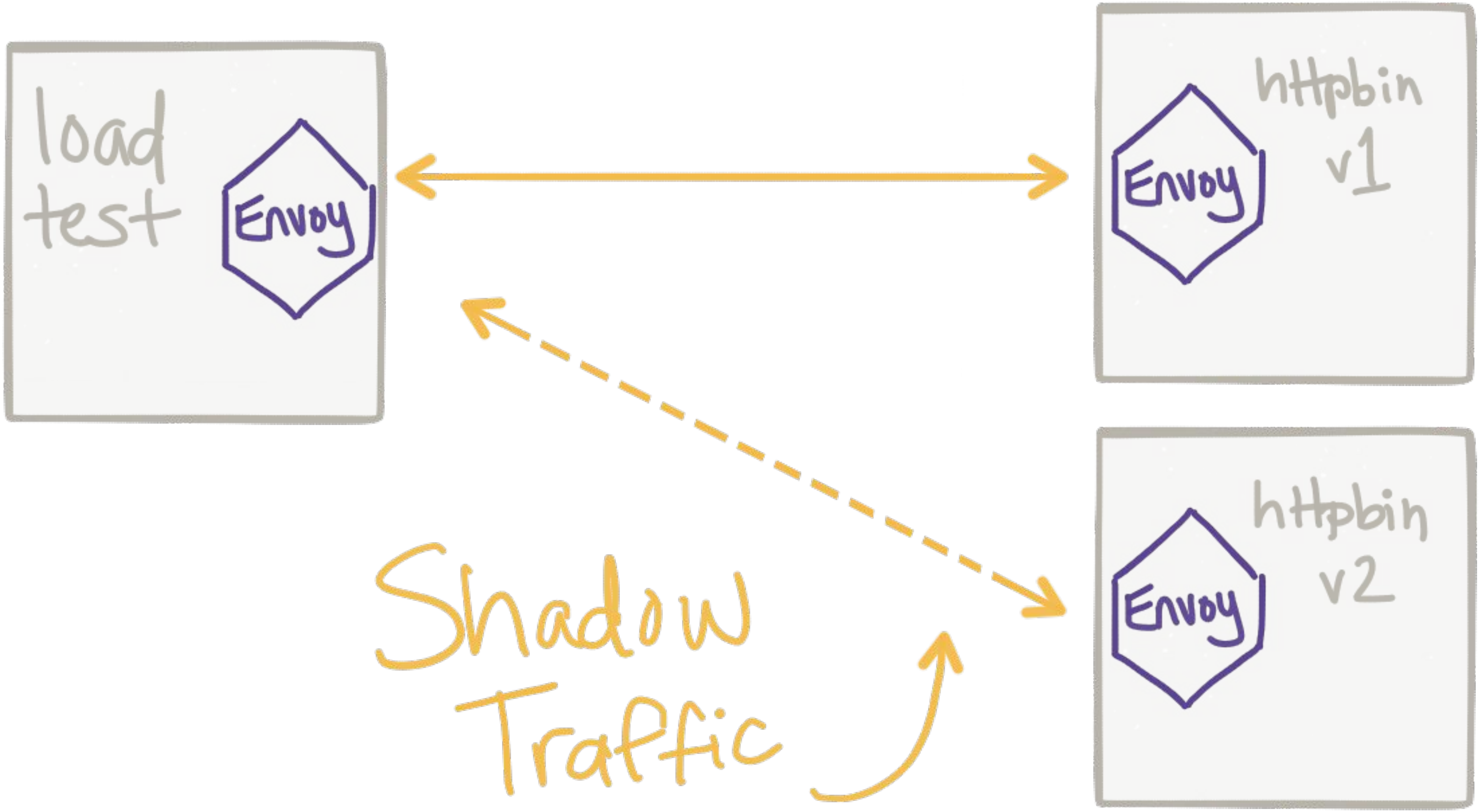




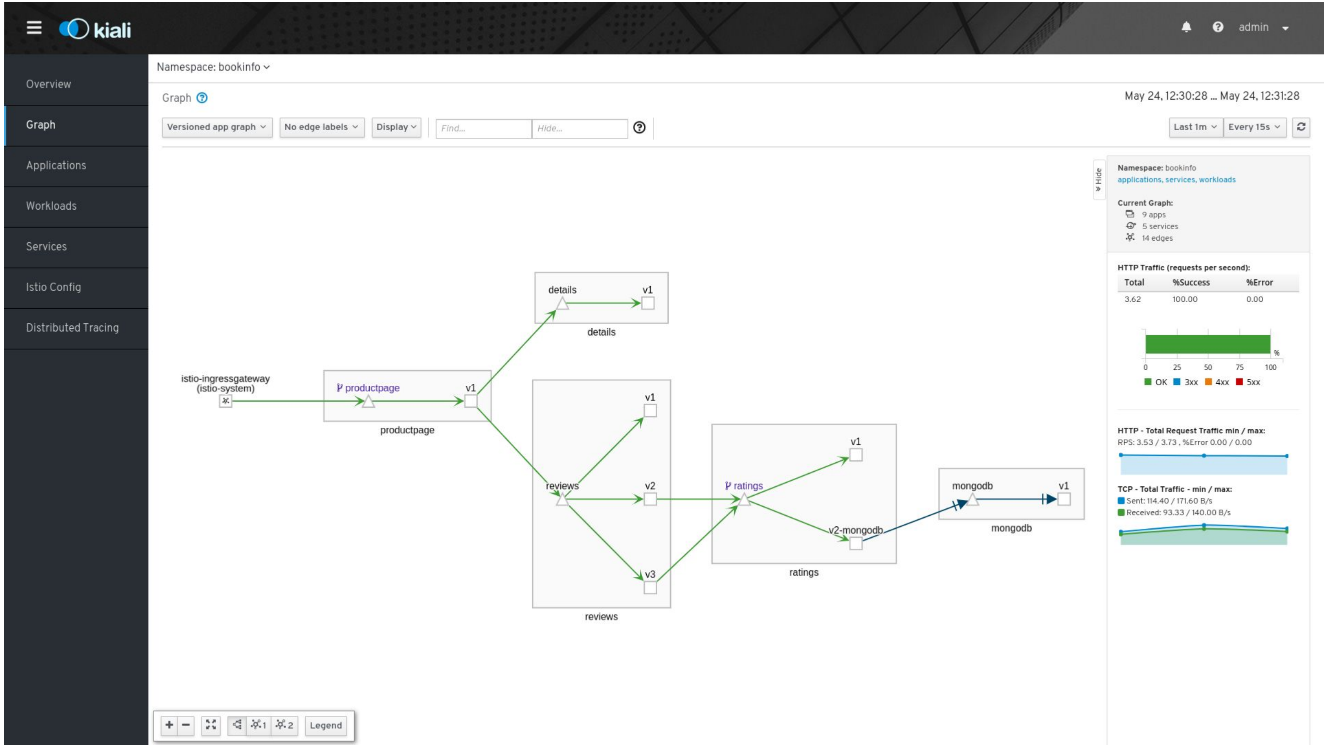
Traffic splitting decoupled from infrastructure scaling - proportion of traffic routed to a version is independent of number of instances supporting the version



Content-based traffic steering - The content of a request can be used to determine the destination of a request


load test
Envoy
httpbin v1
Envoy
httpbin v2
Envoy
Shadow Traffic

Kiali – service mesh observability



Namespace: bookinfo

Graph

May 24, 12:30:28 ... May 24, 12:31:28

Versioned app graph · No edge labels · Display · Find... · Hide...

Last 1m · Every 15s

Namespace: bookinfo
applications, services, workloads

Current Graph:
9 apps
5 services
14 edges

HTTP Traffic (requests per second):

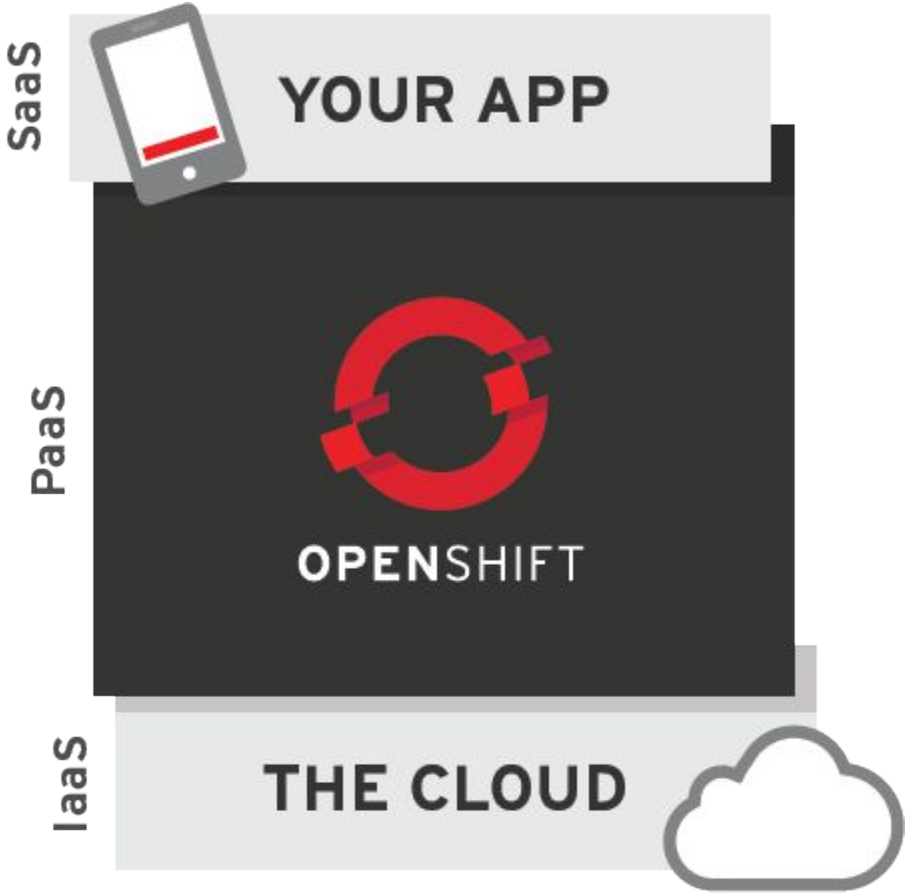
Total	%Success	%Error
3.62	100.00	0.00

HTTP - Total Request Traffic min / max:
RPS: 3.53 / 3.73 , %Error 0.00 / 0.00

TCP - Total Traffic - min / max:
🟦 Sent: 114.40 / 171.60 B/s
🟩 Received: 93.33 / 140.00 B/s

Cloud computing

- IaaS - Infrastructure as a service
  - VMs, servers, storage, network
- PaaS - Platform as a service
  - Execution runtime, database, application server managed Kubernetes, Openshift
- SaaS - Software as a Service
  - Provided applications, CRM, Email, communication



SaaS

YOUR APP

PaaS

OPENSHIFT

IaaS

THE CLOUD

Cloud-native applications

- Basically microservices
- Designed for cloud deployments
- High requirements on
  - **Low memory utilization**
  - **Low processing requirements**
  - **Fast start-ups**
  - Automation of the app lifecycle
  - CI/CD pipelines

12-factor applications

1. **Codebase** - One codebase tracked in revision control, many deploys
2. **Dependencies** - Explicitly declare and isolate dependencies
3. **Config** - Store config in the environment
4. **Backing services** - Treat backing services as attached resources
5. **Build, release, run** - Strictly separate build and run stages

12-factor applications

6. **Processes** - Execute the app as one or more stateles processes
7. **Port binding** - Export services via port binding
8. **Concurrency** - Scale out via the process model
9. **Disposability** - Maximize robustness with fast startup and graceful shutdown
10. **Dev/prod parity** - Keep development, staging, and production as similar as possible

12-factor applications

11. **Logs** - Treat logs as event streams
12. **Admin processes** - Run admin/management tasks as one-off processes

https://12factor.net/

Relational DBs

ORACLE DATABASE

PostgreSQL

Microsoft SQL Server

TERADATA

MariaDB

MySQL

IBM DB2

SYBASE

H2

Access

Apache Derby

HIVE

SQLite

HyperSQL

Distributed tracing

JAEGER

ZIPKIN

Fault tolerance

HYSTRIX
DEFEND YOUR APP

Failsafe

NoSQL DBs

APACHE HBASE

Cassandra

riak

CouchDB relax

mongoDB

HYPERTABLE INC

Neo4j

redis

Security

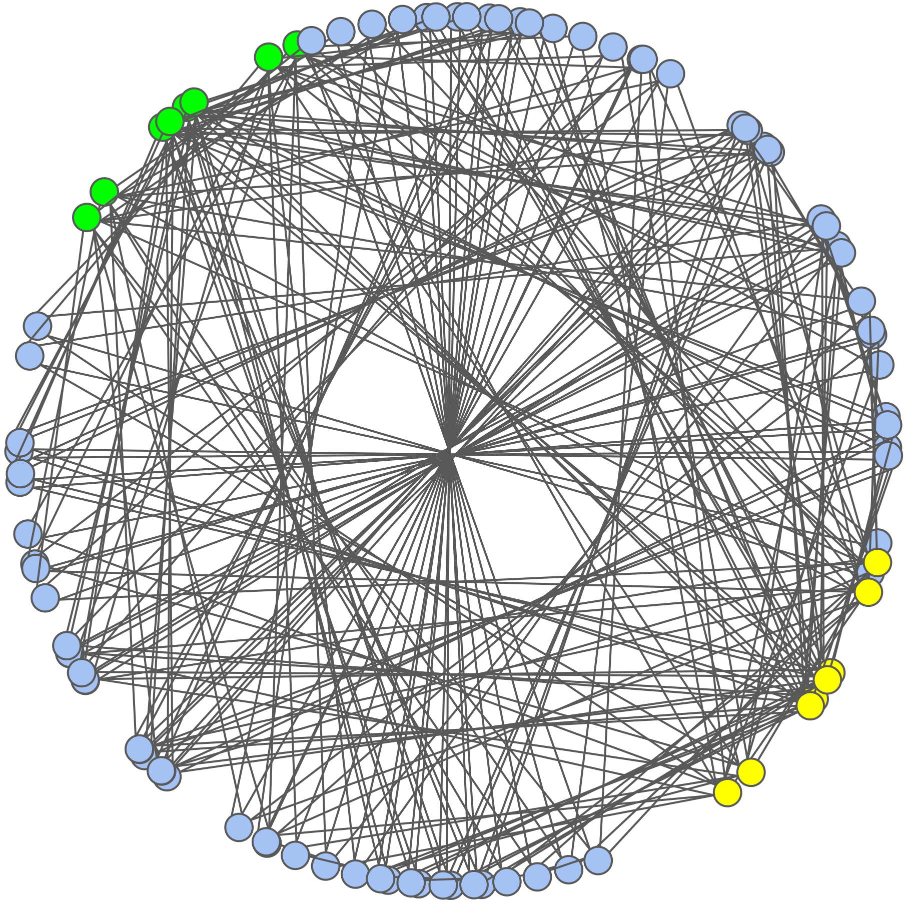
KEYCLOAK

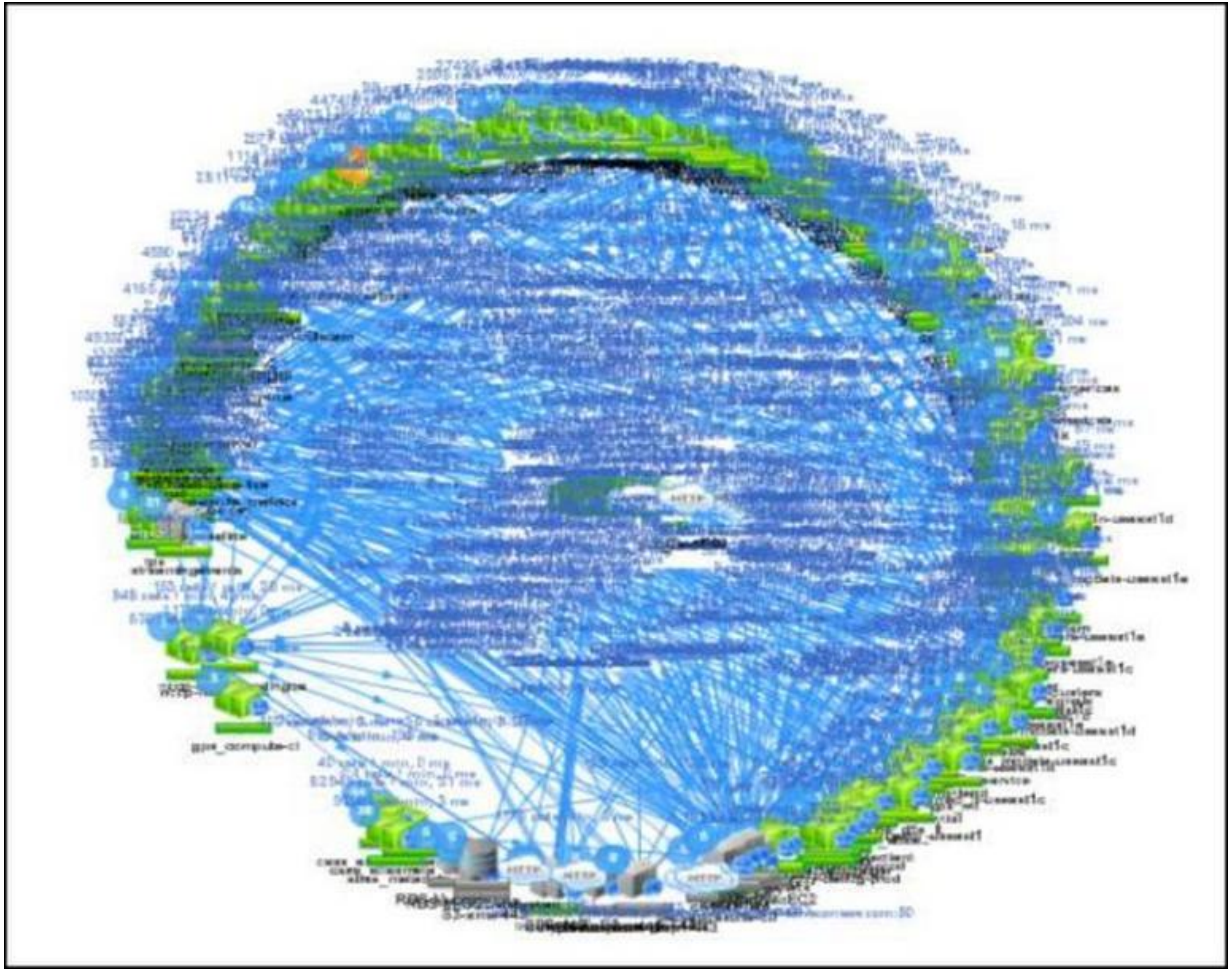
Metrics / monitoring

Prometheus

Grafana

Demo





Thank you

- @xstefank
- xstefank
- xstefank122@gmail.com

Resources

- https://www.zdnet.com/article/to-be-a-microservice-how-smaller-parts-of-bigger-applications-could-remake-it/ originally by Bruce Wong
- https://medium.com/refraction-tech-everything/how-netflix-works-the-hugely-simplified-complex-stuff-that-happens-every-time-you-hit-play-3a40c9be254b
- https://dzone.com/articles/microservices-vs-soa-whats-the-difference
- https://martinfowler.com/articles/microservices.html
- https://www.docker.com/resources/what-container
- https://github.com/kubernetes/kubernetes/blob/master/logo/logo.svg
- https://docs.aws.amazon.com/eks/latest/userguide/dashboard-tutorial.html
- https://www.slideshare.net/asotobu/service-mesh-patterns
- https://access.redhat.com/documentation/en-us/openshift_container_platform/3.3/html/release_notes/release-notes-ocp-3-3-release-notes
- https://thenewstack.io/history-service-mesh/
- https://philcalcado.com/2017/08/03/pattern_service_mesh.html
- https://istio.io/docs/concepts/what-is-istio/
- http://dougbtv.com/nfvpe/2017/06/05/istio-deploy/
- https://blog.aquasec.com/istio-service-mesh-traffic-control
- https://blog.christianposta.com/microservices/traffic-shadowing-with-istio-reduce-the-risk-of-code-release/
- https://github.com/kiali/kiali
- https://blog.openshift.com/what-is-platform-as-a-service-paas/
- https://serverless.zone/abstracting-the-back-end-with-faas-e5e80e837362
- https://softwareengineeringdaily.com/2016/09/08/relational-databases-with-craig-kerstiens/
- https://www.getfilecloud.com/blog/2014/08/leading-nosql-databases-to-consider/
- https://www.jaegertracing.io/
- https://blog.twitter.com/engineering/en_us/a/2012/distributed-systems-tracing-with-zipkin.html
- https://www.trzcacak.rs/imgm/iTJioJh_prometheus-logo-logo-prometheus/
- https://en.wikipedia.org/wiki/File:Grafana_logo.png
- https://design.jboss.org/keycloak/index.htm
- https://github.com/Netflix/Hystrix

Principles of microservices

Principles of microservices

- Perspective of business use cases and the solution architecture
- Based on the work of Sam Newman
  - S. Newman,Building Microservices, 1st ed. O’Reilly Media, Inc.,2015
  - S. Newman, “Principles of Microservices,” 2016. [Online]. Available: http://samnewman.io/talks/principles-of-micro services/

1. Modeled around the business concepts

- Microservices and teams that are maintaining them should correspond to the same business domain
- Microservices are more stable
- Requirements don’t change frequently
- Developers are focused on the particular system segment

2. Adapting a culture of automation

- With the increasing number of services, their maintenance , administration, and deployment can become unmanageable
- Automation of the is essential
  - Service testing
  - CI/CD
  - Deployment strategy

3. Hiding the internal implementation details

- To keep the option of independent development
- Related to bounded-contexts defined by DDD (domain driven design)
    - The context is separated by an explicit interface represented as an API
    - Teams can specify which utilities of the service can be shared and which must be hidden
    - Every request must be processed through this interface

4. Decentralizating all things

- Self-sustaining development == services are maintained autonomously
- Decision making delegated to the team maintaining the microservice
- Relevant business logic should be kept in the service and the communication should be as simple as possible

Conway’s law

Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization’s communication structure.

5. Independent deployments

- Most important principle
- A microservice deployment shouldn't influence the lifespan of any other service
- Various techniques
  - Consumer-driven contracts
  - Test suites for individual parts of the domain
  - CI
  - B/G testing, canary deployments, mirroring

6. Customer first

- Service calls must be as simple as possible for customers
- API documentation (Swagger, OpenAPI)
- Service discovery
- Transparency of the call propagation

7. Failure isolation

- Fault tolerance to other services failures
- No single point of failure
- Network failures
- Various techniques
  - Fallbacks
  - Retries
  - Timeouts
  - Circuit breakers
  - Bulkheads

8. High observability

- Monitoring
  - Individual services
  - System monitoring
- Tracing
  - Synthetic transactions
  - Correlation IDs
- Logging
  - Aggregated logs