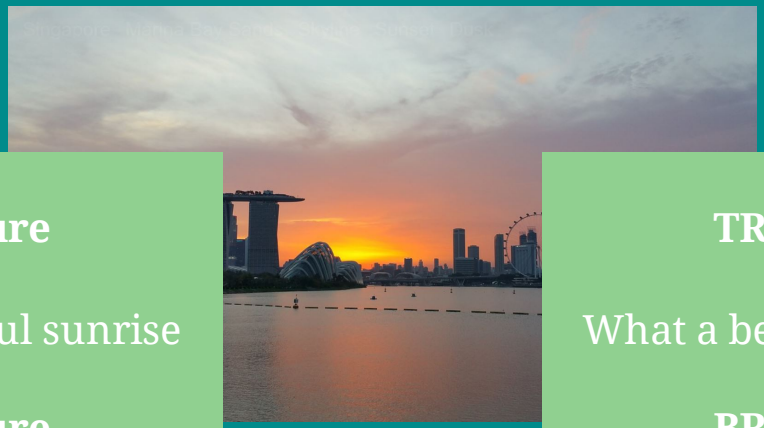
From Monolith to Micro-services with Kubernetes

16 Mar 2019, FOSS Asia, Singapore

Michael Bright, @mjbright

Slides & source code at https://mjbright.github.io/Talks

TEST PAGE



TL-Nature

What a beautiful sunrise

TR-Nature

What a beautiful sunrise

BL-Nature

What a beautiful sunrise

CM-Nature

What a beautiful sunrise

What a beautiful sunrise

Michael Bright, @mjbright

Freelance Trainer: Kubernetes, Serverless, Docker, CloudNative

Past researcher, dev, team lead, dev advocate

British, living in France for 27-years

Docker Community Lead, Python User Group

linkedin.com/in/mjbright github.com/mjbright

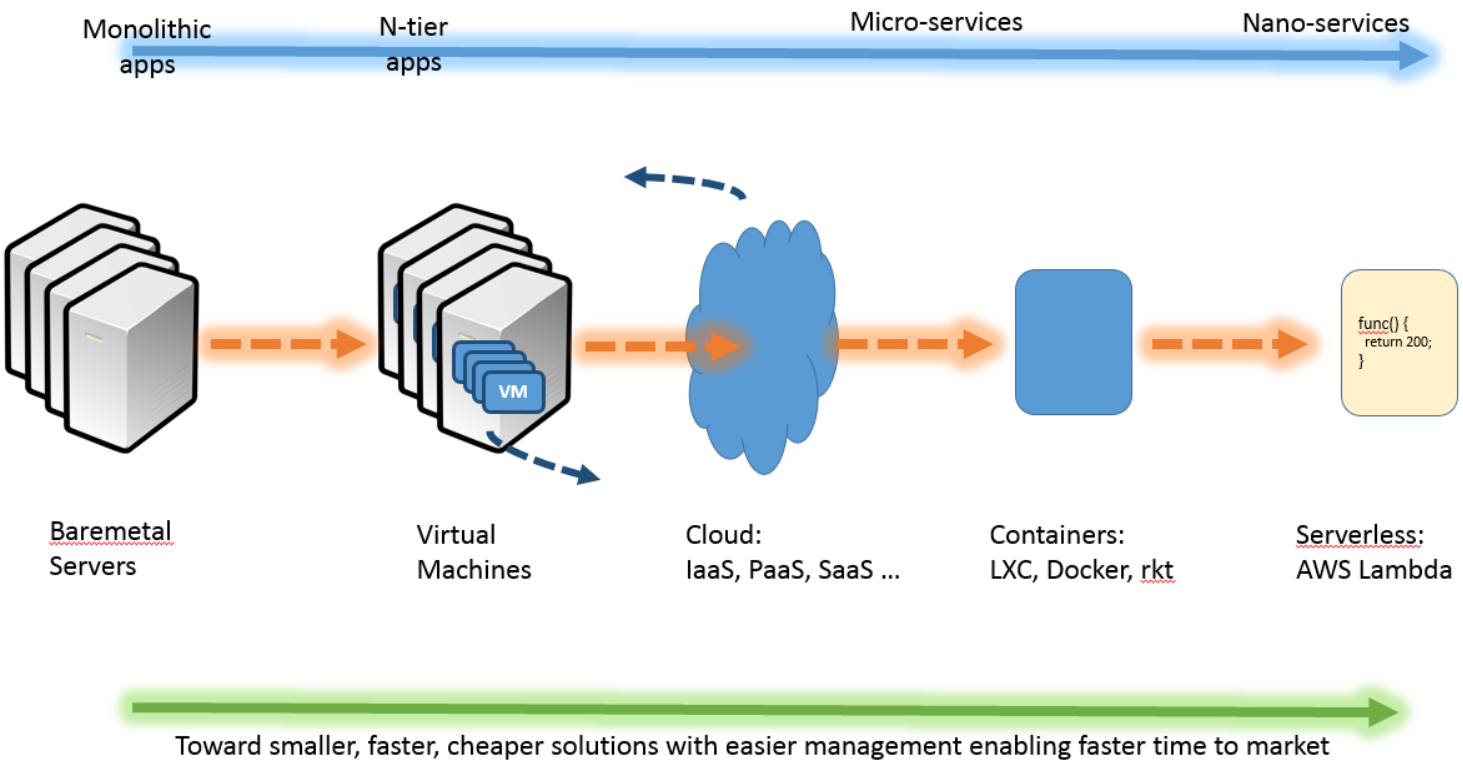
Outline

- Monoliths to Micro-services
- Orchestration: Kubernetes
- Deployment Strategies
- Architecture Design patterns
- Summary

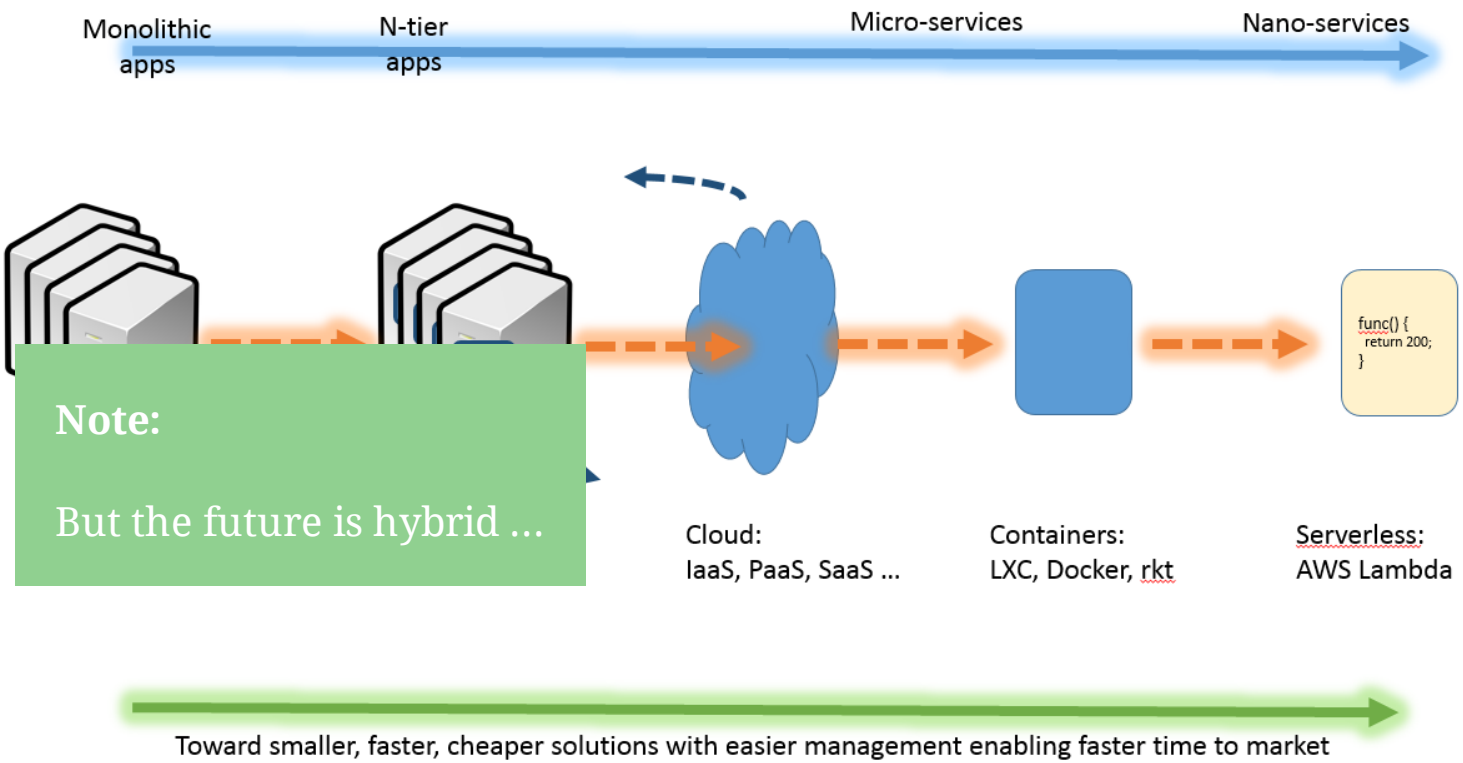
Outline

- Monoliths to Micro-services
  - Orchestration: Kubernetes
  - Deployment Strategies
  - Architecture Design patterns
  - Summary

First ... a bit of history

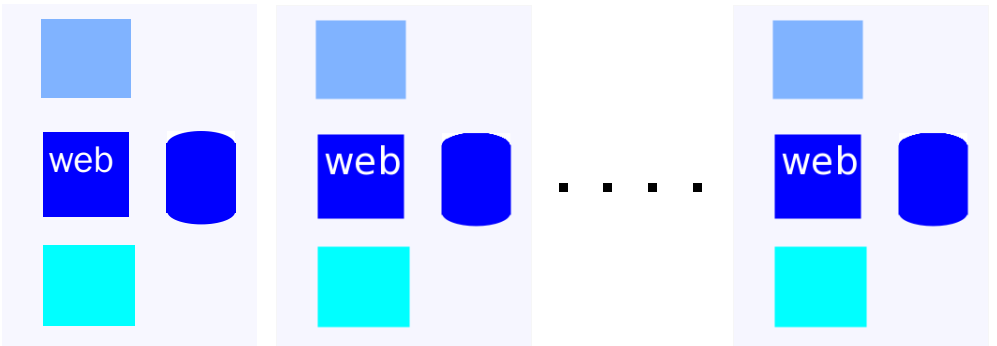


First ... a bit of history



Monoliths to Micro-services

Monoliths are **deployed, scaled, upgraded, reimplemented** as complete units



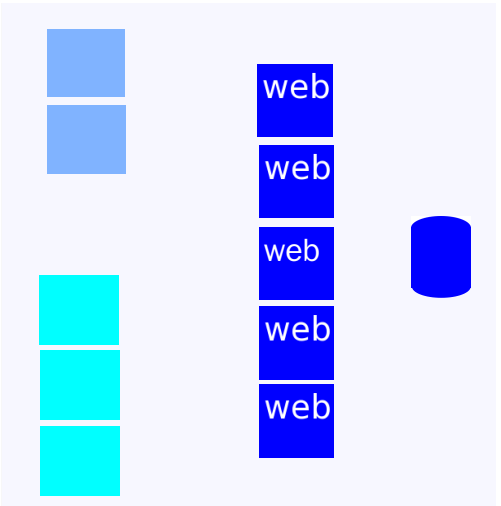
Problem: Feel the pain

- components are tightly coupled and only scale as a unit.
- are developed in Waterfall
- difficult to patch
- library dependencies between components
- difficult to reuse

Monoliths to Micro-services

Proposition: split up components

Individual μ-service components can be **deployed, scaled, upgraded, reimplemented** ...



In micro-service architecture components are only lightly coupled

- interconnected by network
- can be scaled independently
- can be deployed/upgraded independently

Advantages of Micro-services

Separation of Concerns - "do one thing well"

Advantages of Micro-services

Separation of Concerns - "do one thing well"

Smaller Projects/teams

Advantages of Micro-services

Separation of Concerns - "do one thing well"

Smaller Projects/teams

Ease Scaling, Deployment, Testing, Evolution

Advantages of Micro-services

Separation of Concerns - "do one thing well"

Smaller Projects/teams

Ease Scaling, Deployment, Testing, Evolution

Loosely coupled components

Advantages of Micro-services

Separation of Concerns - "do one thing well"

Smaller Projects/teams

Ease Scaling, Deployment, Testing, Evolution

Loosely coupled components

Allow for composition of new services

Can be re-implemented

Advantages of Micro-services

Separation of Concerns - "do one thing well"

Smaller Projects/teams

Ease Scaling, Deployment, Testing, Evolution

Loosely coupled components

Allow for composition of new services

Can be re-implemented

So are they a panacea?

Disadvantages

Greater complexity

- Requires orchestration, component version management
- Greater organizational complexity
- Monitoring, debugging, end-2-end test are more difficult

Disadvantages

Greater complexity

- Requires orchestration, component version management
- Greater organizational complexity
- Monitoring, debugging, end-2-end test are more difficult

More network communication

- Network error handling, Performance, Circuit-breakers

Disadvantages

Greater complexity

- Requires orchestration, component version management
- Greater organizational complexity
- Monitoring, debugging, end-2-end test are more difficult

More network communication

- Network error handling, Performance, Circuit-breakers

Still requires best practices

- Behaviour and Test-Driven Development, CI/CD
- Documentation of interfaces/APIs, Stable

Outline

- Monoliths to Micro-services
- Orchestration: Kubernetes
- Deployment Strategies
- Architecture Design patterns
- Summary

Orchestration: Kubernetes

Problem: Feel the pain Impossible to manage 10000 containers running across a data center of 1000 nodes.

- on which nodes should you schedule
- which contianers are malfunctioning
- which are started and ready to go ==>

We need Orchestration

REPLACE: As you move from deploying containers on a single machine to deploying them across a number of machines, you'll need an orchestration tool to manage (and automate) the arrangement, coordination, and availability of the containers across the entire system.

REPLACE: Issues:

- Cross-server container communication
- Horizontal scaling
- Service discovery
- Security/TLS
- Zero-downtime deploys
- Rollbacks

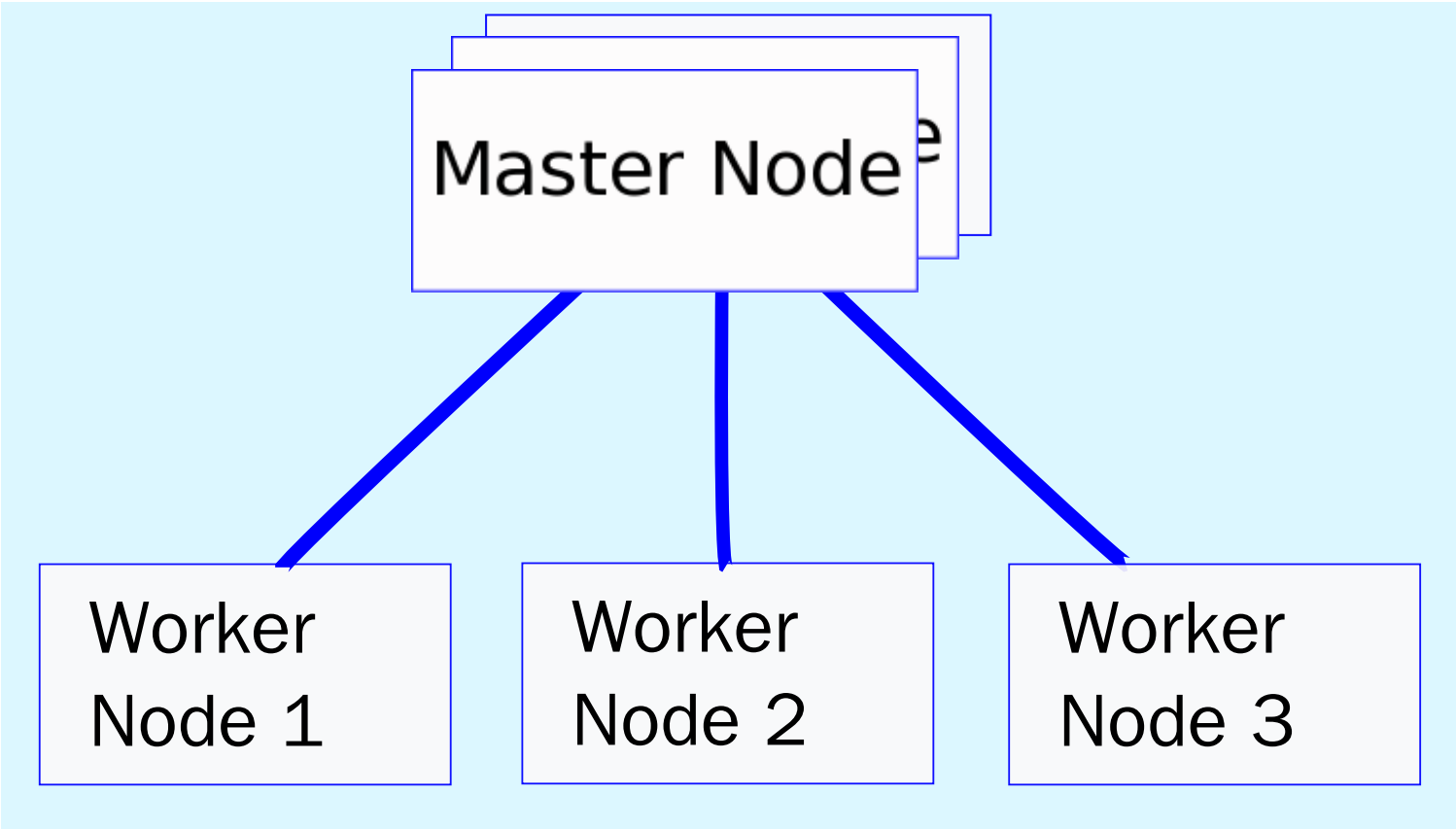
REPLACE: Orchestration Feature Wish-list

Feature Info Health checks Verify when a task is ready to accept traffic Dynamic port-mapping Ports are assigned dynamically when a new container is spun up Zero-downtime deployments Deployments do not disrupt end users Service discovery Automatic detection of new containers and services Auto scaling Automatically scale resources up or down based on the load Provisioning New containers should select hosts based on resources and configuration

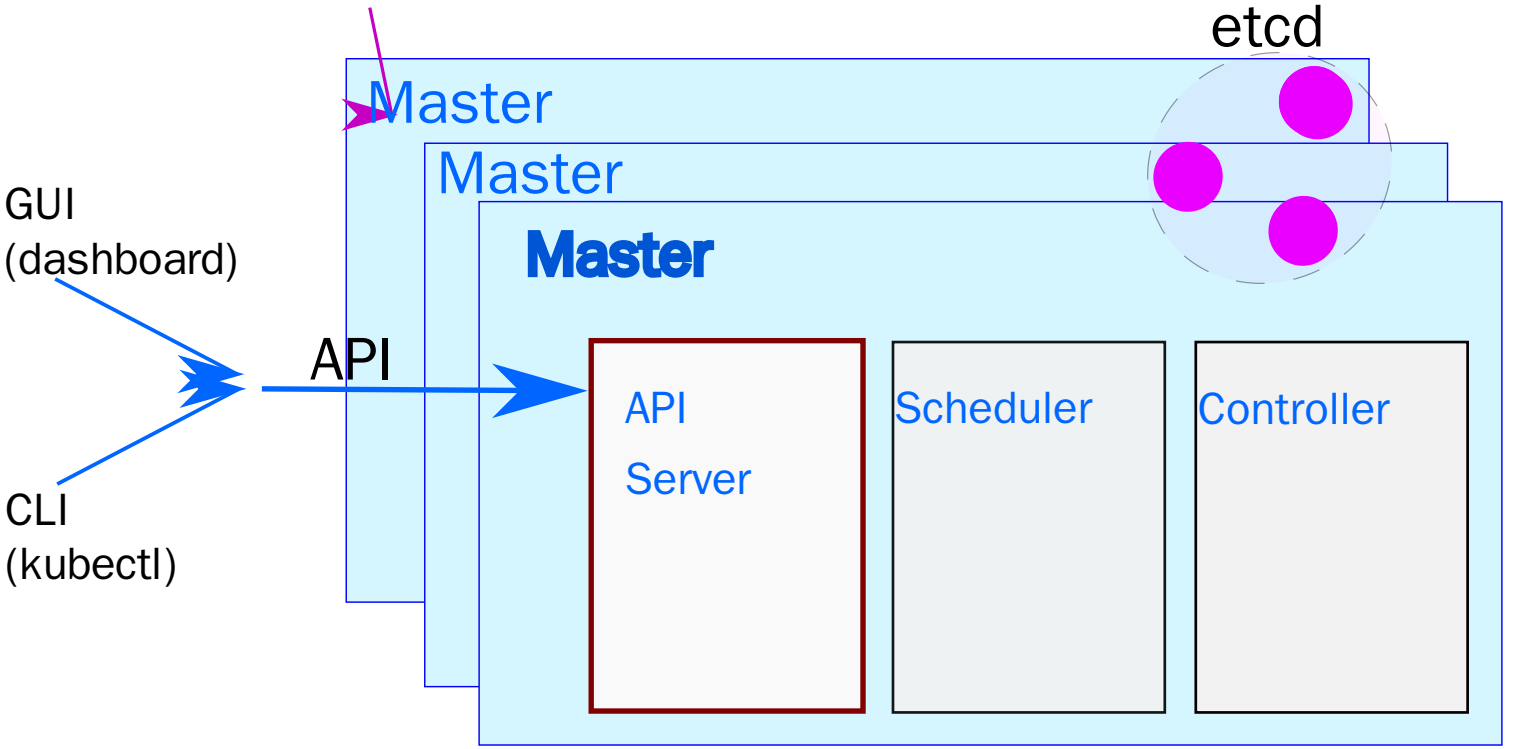
REPLACE: What else?

Load balancing, logging, monitoring, authentication and authorization, security… predictability, scalability, and high availability…

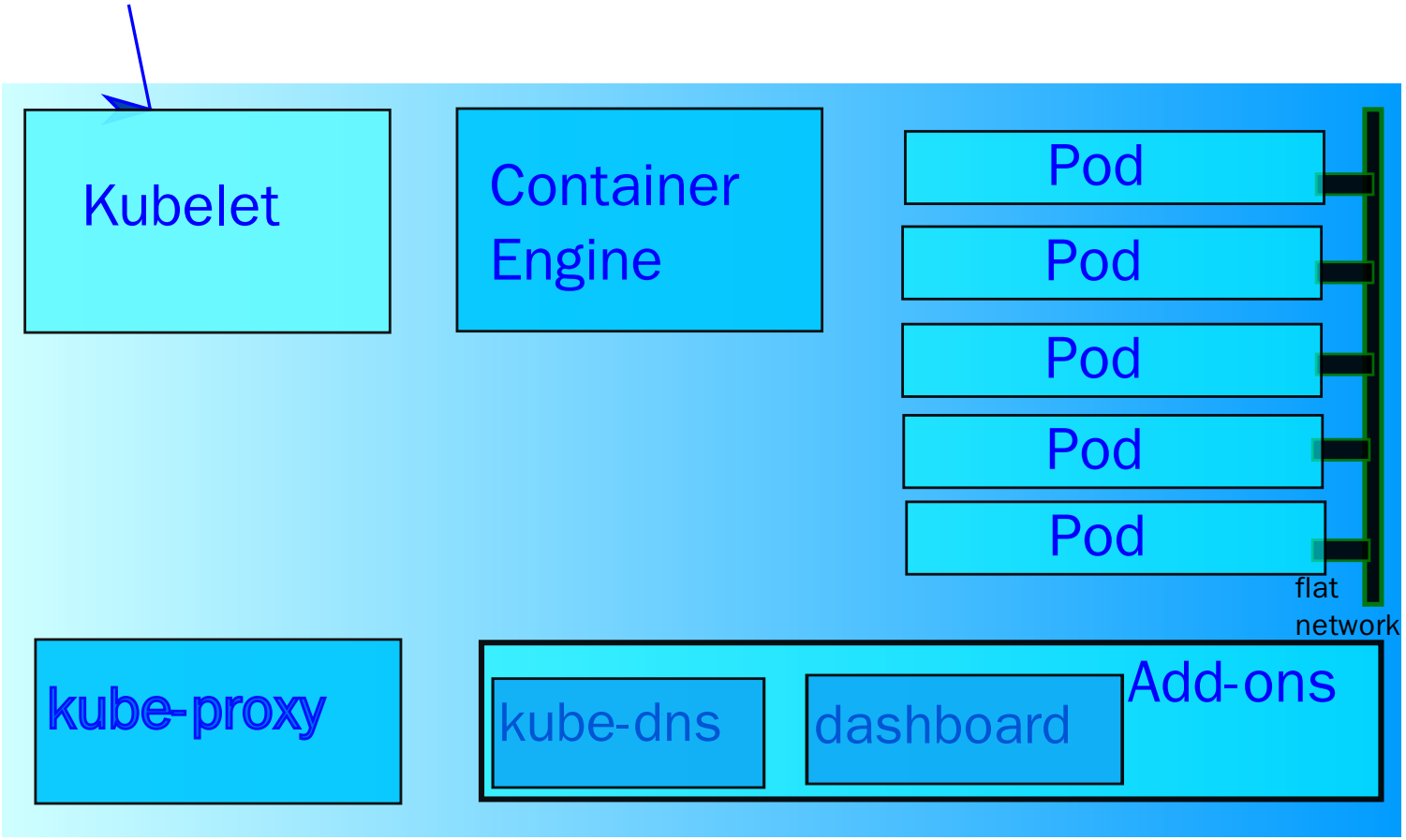
Kubernetes - Architecture



Kubernetes - Master Nodes

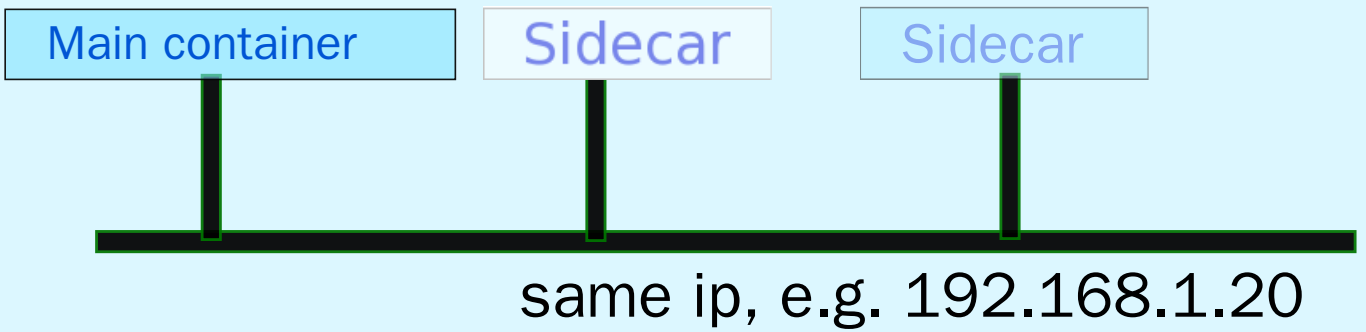


Kubernetes - Worker Nodes



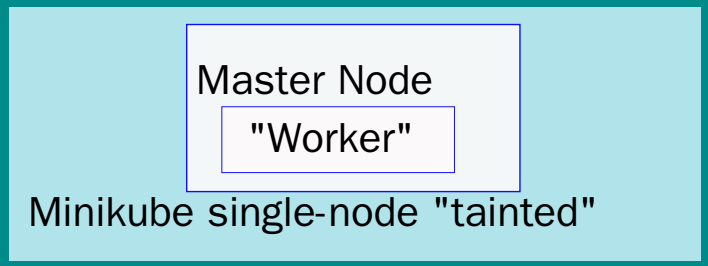
Kubernetes - Pods

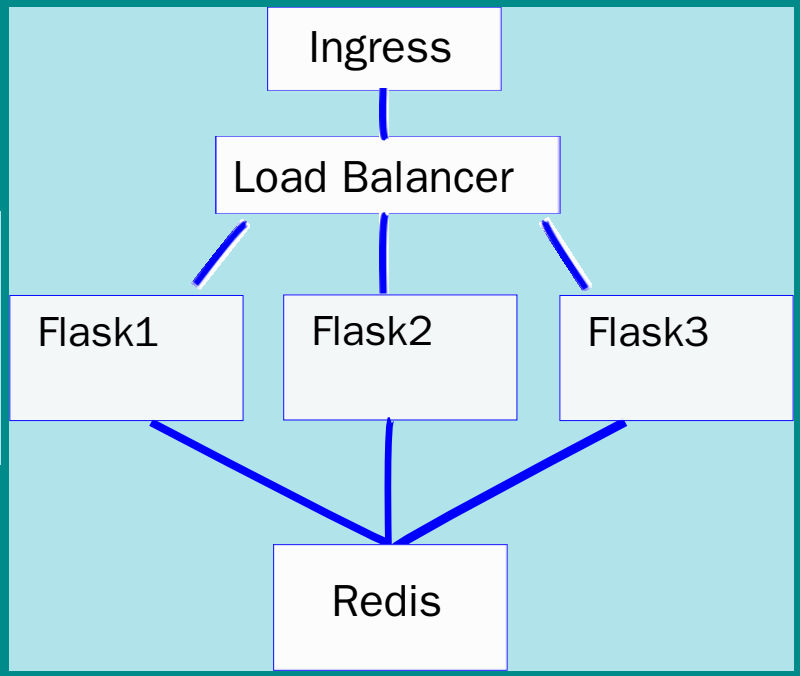
Containers share some namespaces:
- PID, IPC, network , time sharing



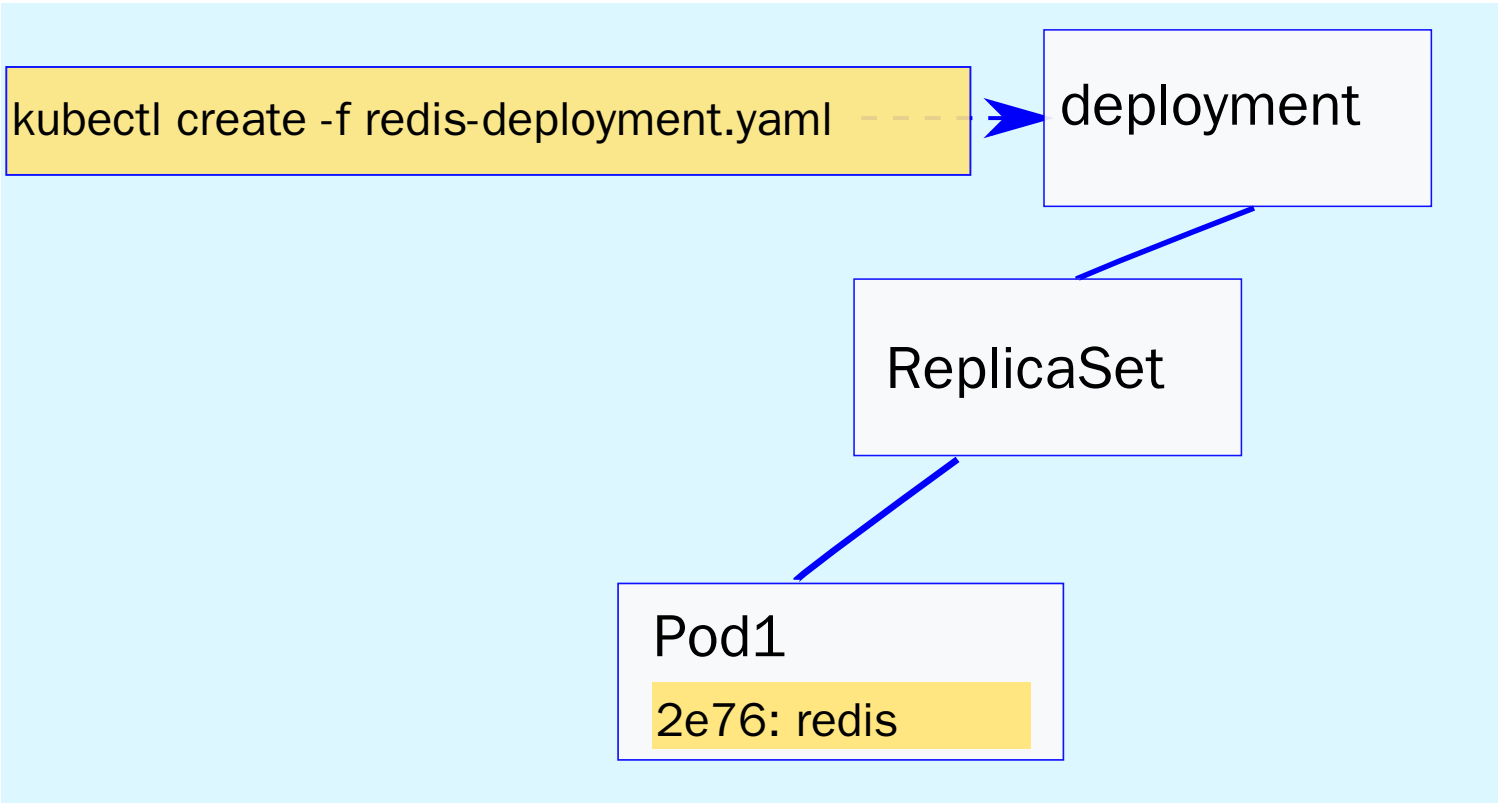
A pod houses one or more containers

Kubernetes Demo





Kubernetes - Deploying Redis



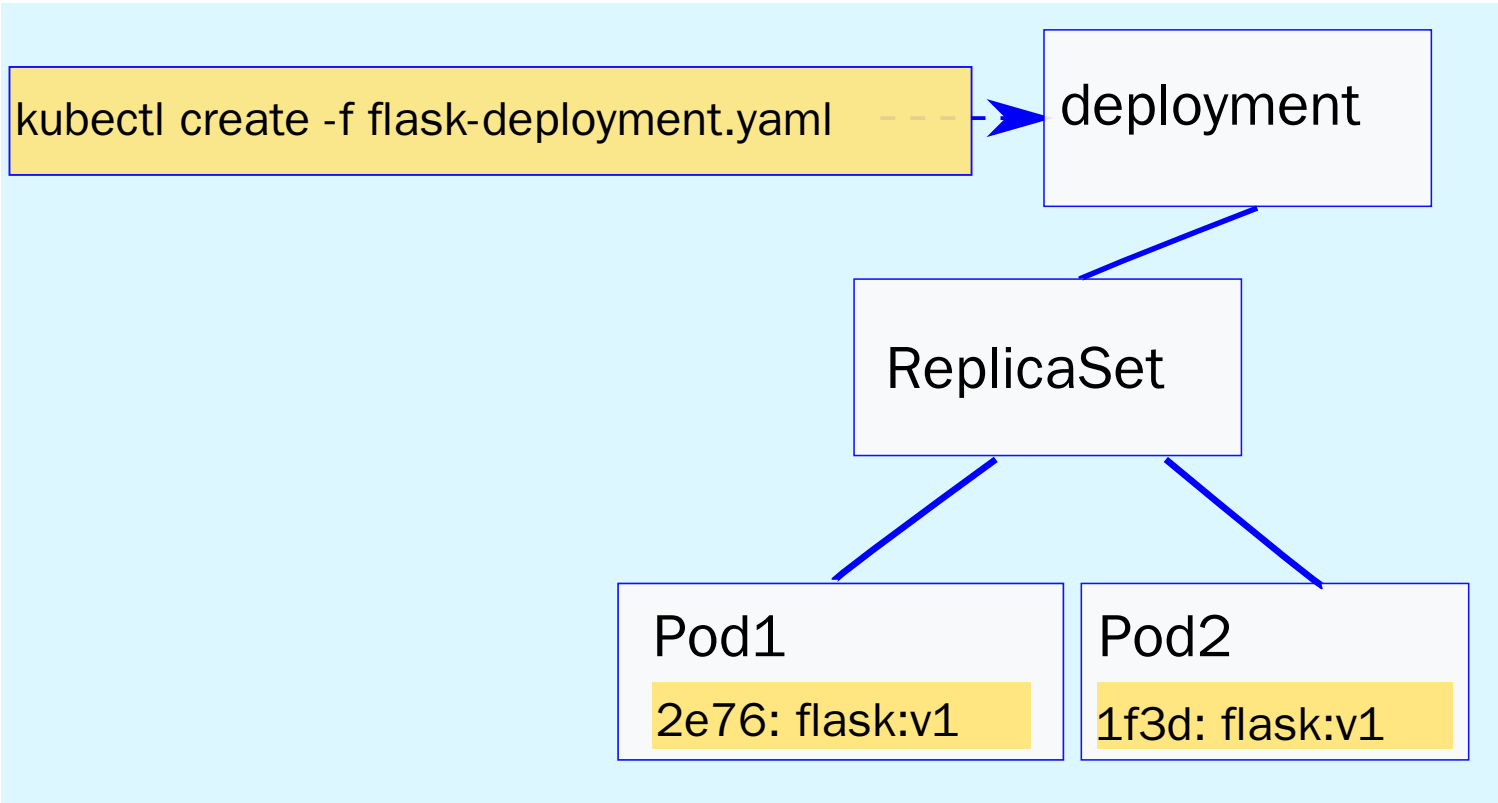
Kubernetes - Deploying Redis

```
# kubectl run redis --image=redis:latest --port=6379

$ kubectl apply -f redis-deployment.yaml
deployment.extensions "redis" created

$ kubectl get pods
NAME                     READY     STATUS              RESTARTS   AGE
redis-68595c4d95-rr4pr   0/1       ContainerCreating   0          1s
```

Kubernetes - Deploying Redis (yaml)

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  labels:
    run: redis
  name: redis
spec:
  replicas: 1
  selector:
    matchLabels:
      run: redis
  template:
    metadata:
      labels:
        run: redis
    spec:
      containers:
      - image: redis:latest
        name: redis
        ports:
        - containerPort: 6379
```

Kubernetes - Deploying Flask



Kubernetes - Deploying Flask

```
# kubectl run flask-app --image=$IMAGE --port=5000

$ kubectl apply -f flask-deployment.yaml
deployment.extensions "flask-app" created

$ kubectl get pods
NAME                         READY     STATUS              RESTARTS   AGE
flask-app-8577b44db-96cht    0/1       Pending             0          1s
redis-68595c4d95-rr4pr       0/1       ContainerCreating   0          1s
```

Kubernetes - Deploying Flask (yaml)

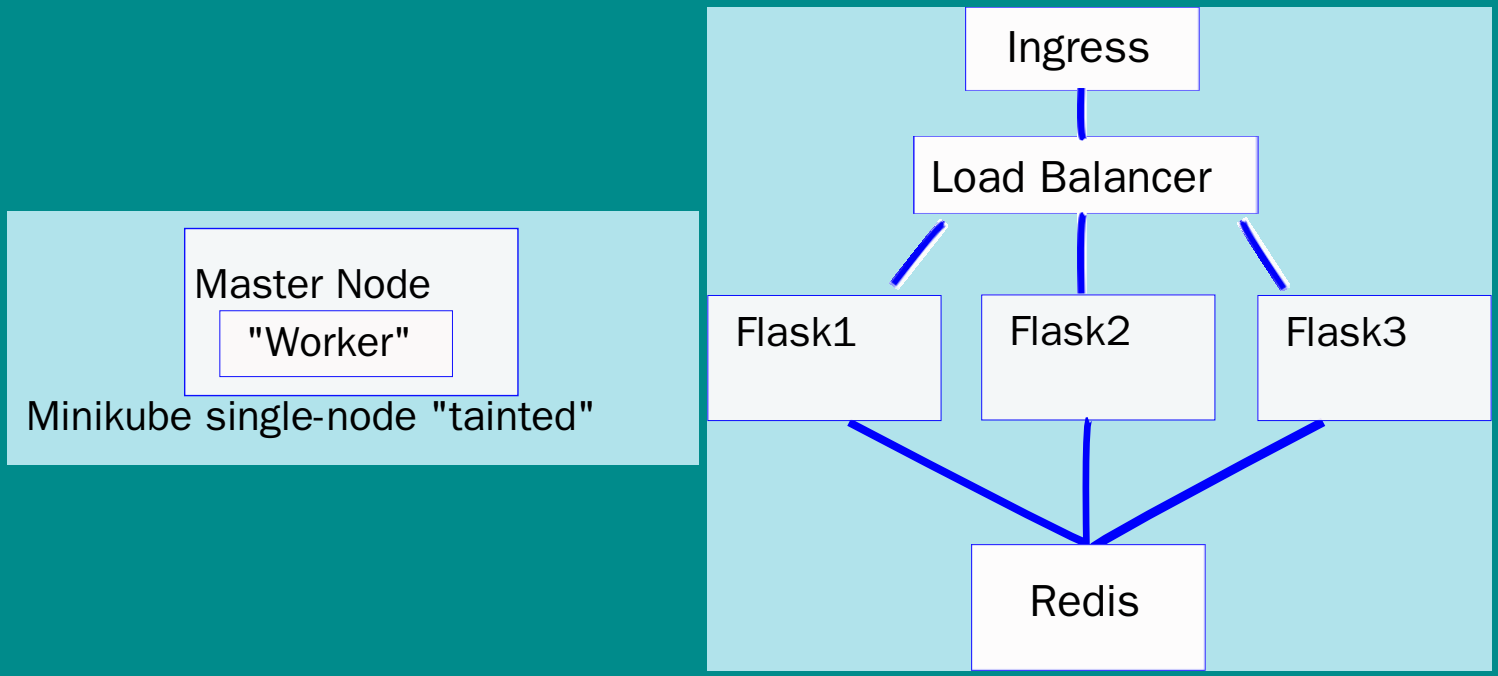
```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  labels:
    run: flask-app
  name: flask-app
spec:
  replicas: 1
  selector:
    matchLabels:
      run: flask-app
  template:
    metadata:
      labels:
        run: flask-app
    spec:
      containers:
      - image: mjbright/flask-web:v1
        name: flask-app
        ports:
        - containerPort: 5000
```

Operations - Scaling

```
# kubectl scale deploy flask-app --replicas=4

$ kubectl edit -f flask-deploy.yaml
```

```
...
spec:
  replicas: 4
```

Demo


Master Node
"Worker"
Minikube single-node "tainted"
Ingress
Load Balancer
Flask1
Flask2
Flask3
Redis

Outline

- Monoliths to Micro-services
- Orchestration: Kubernetes
- Deployment Strategies
- Architecture Design patterns
- Summary

????

Problem: Feel the pain

… so … **Proposition: split up components**

Micro-service Deployment Strategies

- Rolling Upgrade
- Health Checks
- Strangler Pattern

Operations - Rolling Upgrades

Several strategies exist

recreate - terminate old version before releasing new one

Operations - Rolling Upgrades

Several strategies exist

recreate - terminate old version before releasing new one

ramped - gradually release a new version on a rolling update fashion

Operations - Rolling Upgrades

Several strategies exist

recreate - terminate old version before releasing new one

ramped - gradually release a new version on a rolling update fashion

blue/green - release new version alongside old version then switch

Operations - Rolling Upgrades

Several strategies exist

recreate - terminate old version before releasing new one

ramped - gradually release a new version on a rolling update fashion

blue/green - release new version alongside old version then switch

canary - release new version to subset of users, proceed to full rollout

Operations - Rolling Upgrades

Several strategies exist

recreate - terminate old version before releasing new one

ramped - gradually release a new version on a rolling update fashion

blue/green - release new version alongside old version then switch

canary - release new version to subset of users, proceed to full rollout

a/b testing - release new version to subset of users in a precise way (HTTP headers, cookie, weight, etc.).

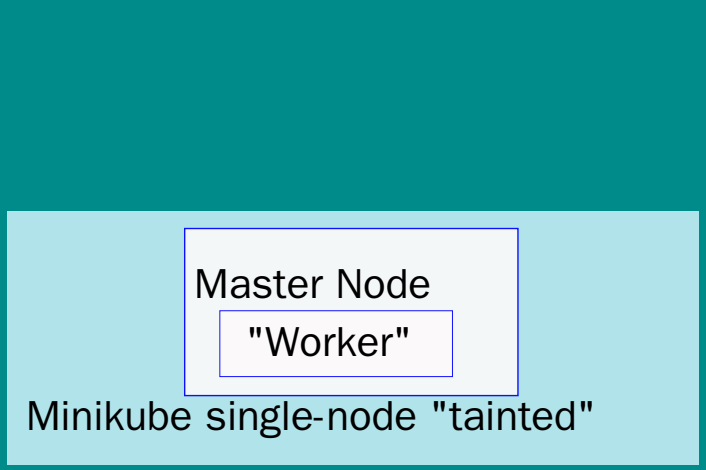
Operations - Rolling Upgrade

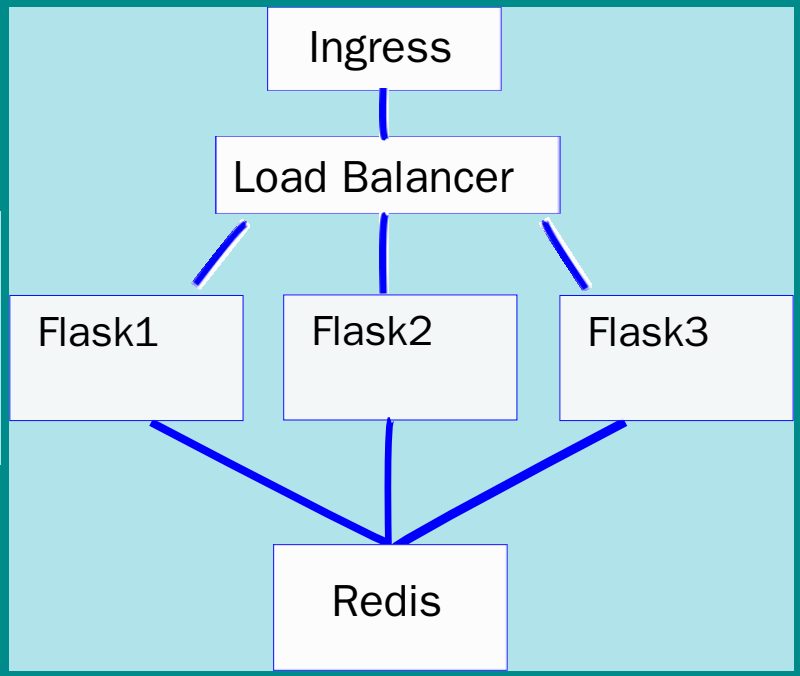
Ramped

```
# kubectl set image deploy flask-app flask-app=mjbright/flask-web:v2

$ kubectl edit -f flask-deploy.yaml
$ kubectl rollout status deployment/flask-app
```

```
...
    spec:
      containers:
      - image: mjbright/flask-web:v2
```

Demo


Master Node
"Worker"
Minikube single-node "tainted"


Ingress
Load Balancer
Flask1
Flask2
Flask3
Redis

????

Problem: Feel the pain

… so … **Proposition: split up components**

Operations - Healthchecks

Cover healthchecks

and Readiness probes

????

Problem: Feel the pain

… so … **Proposition: split up components**

Operations - Strangler Pattern

The Strangler is a pattern used in the initial migration from a Monolithic architecture to a Micro-services architecture

Micro-service - Architecture Design Patterns

We are not concerned with:

Standard Component Patterns

Micro-services themselves (!) - Fine-grained SOA

Sidecar

Micro-service - Architecture Design Patterns

We are concerned with:

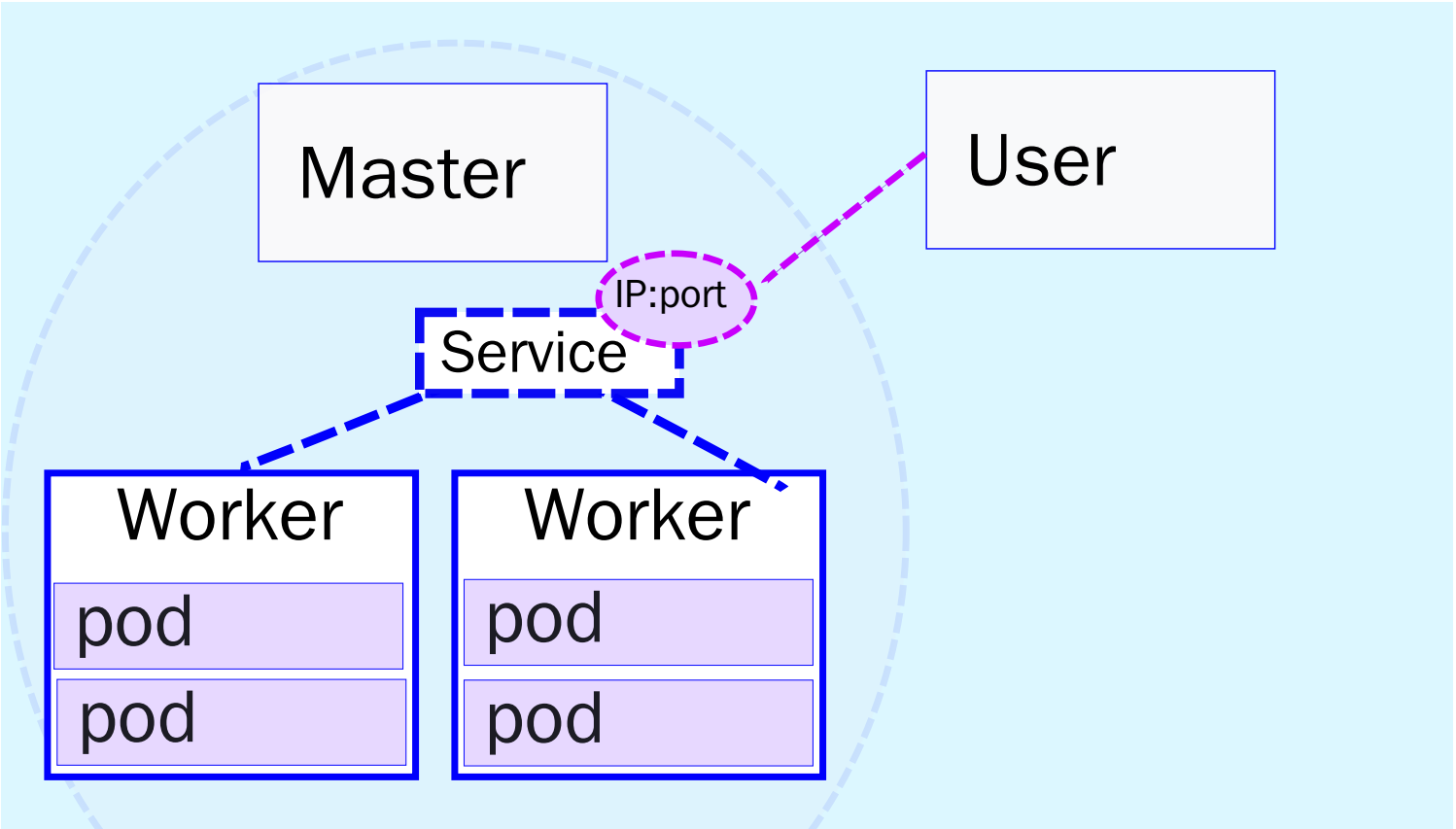
- Exposing Services
- Ingress
- API Gateway
- Service Mesh
- Hybrid Apps

????

Problem: Feel the pain

… so … **Proposition: split up components**

Kubernetes - Exposing Services

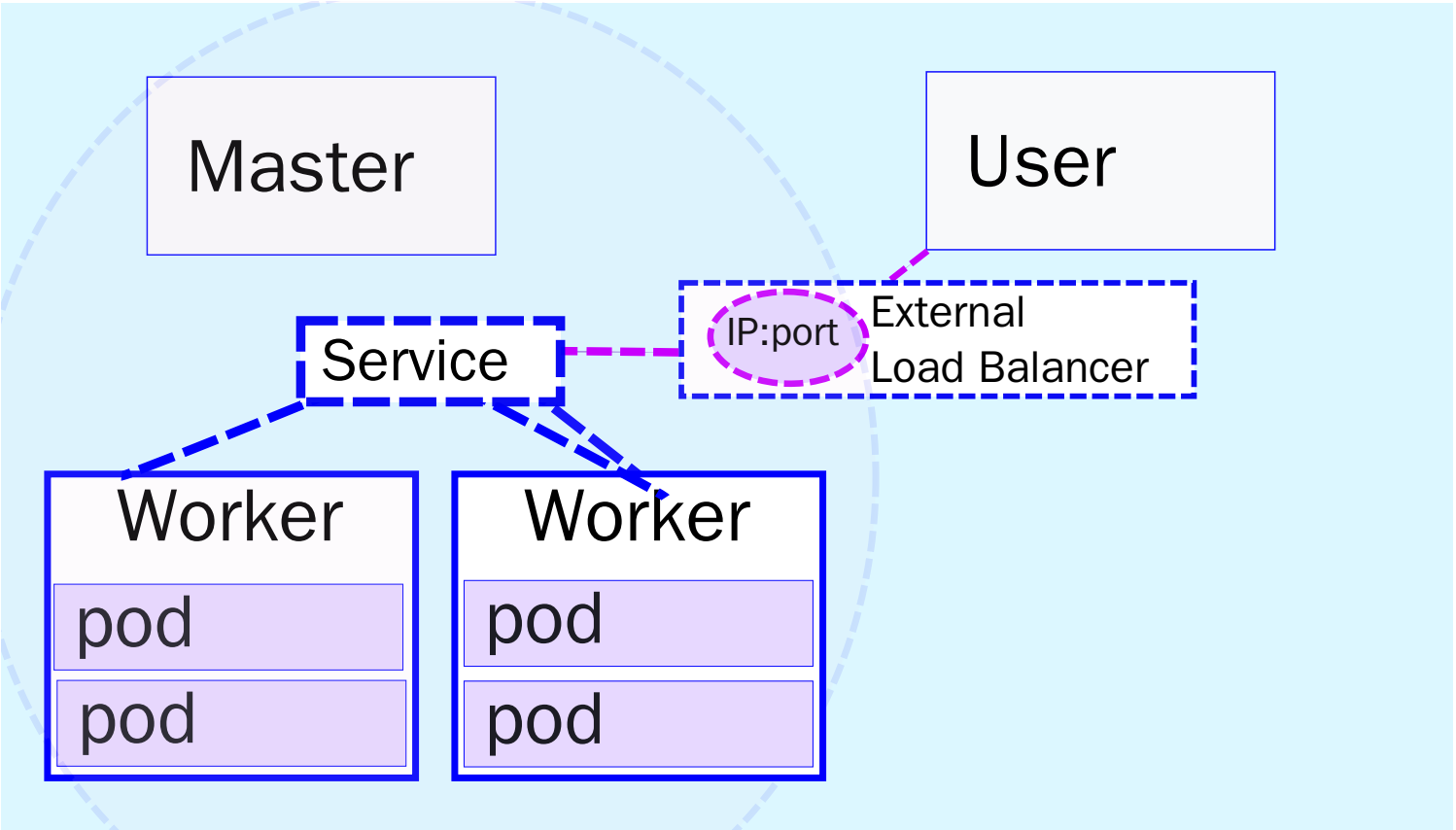


Design Pattern - Services

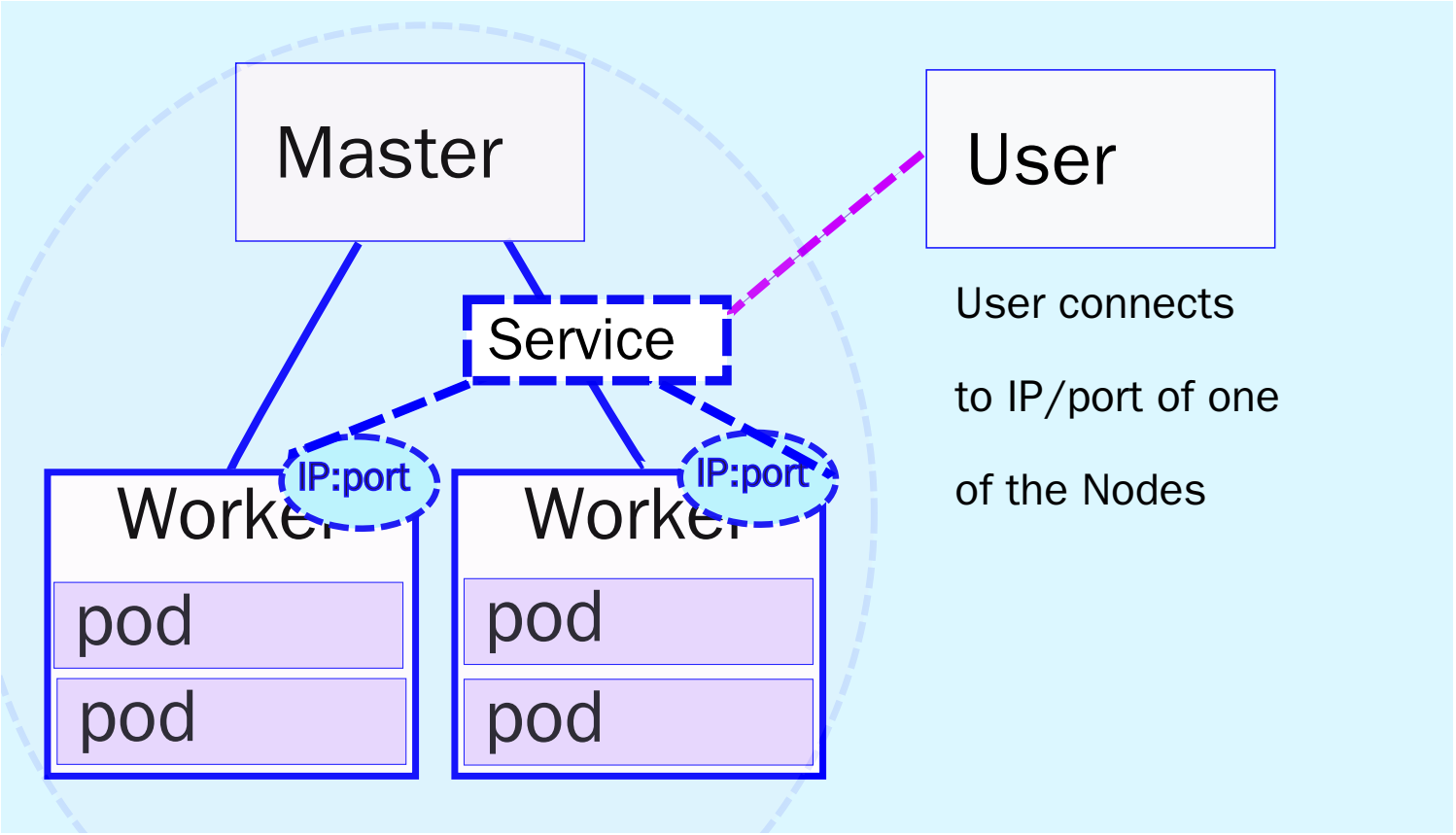
Services can be exposed via

- NodePort
- HostPort
- ClusterIP
- LoadBalancer

Exposing Services (LoadBalancer)



Exposing Services (NodePort)

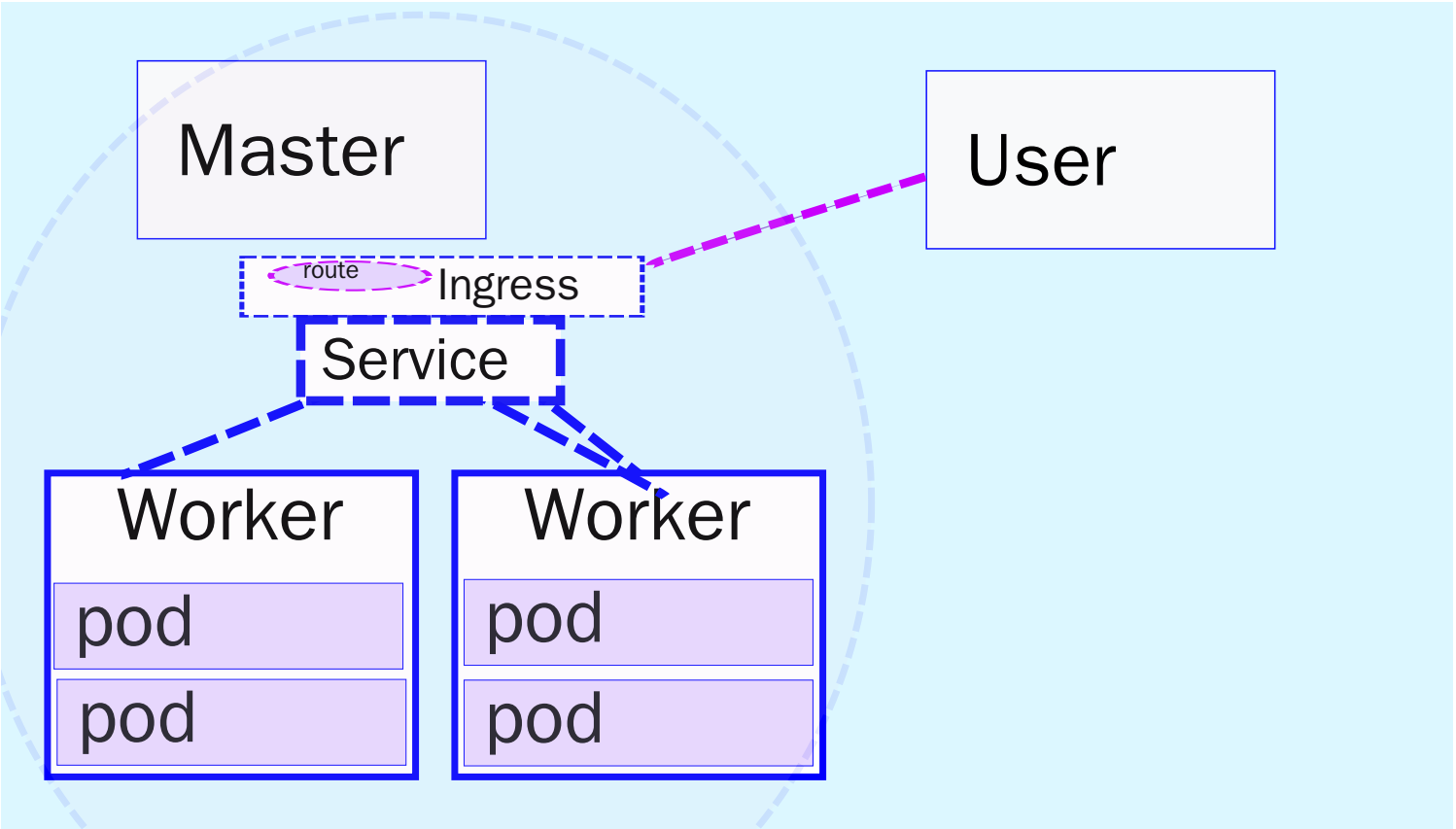


????

Problem: Feel the pain

… so … **Proposition: split up components**

Exposing Services (IngressController)



Exposing Redis Service (LoadBalancer)

```
# kubectl expose deployment redis --type=LoadBalancer

$ kubectl apply -f redis-service.yaml
service "redis" created

$ kubectl get svc
NAME         TYPE           CLUSTER-IP       EXTERNAL-IP   PORT(S)          AGE
kubernetes   ClusterIP      10.96.0.1        <none>        443/TCP          5h
redis        LoadBalancer   10.101.158.201   <pending>     6379:31218/TCP   1s
```

Exposing Redis Service (LoadBalancer)

```
apiVersion: v1
kind: Service
metadata:
  labels:
    run: redis
  name: redis
spec:
  ports:
  - port: 6379
    protocol: TCP
    targetPort: 6379
  selector:
    run: redis
  type: LoadBalancer
```

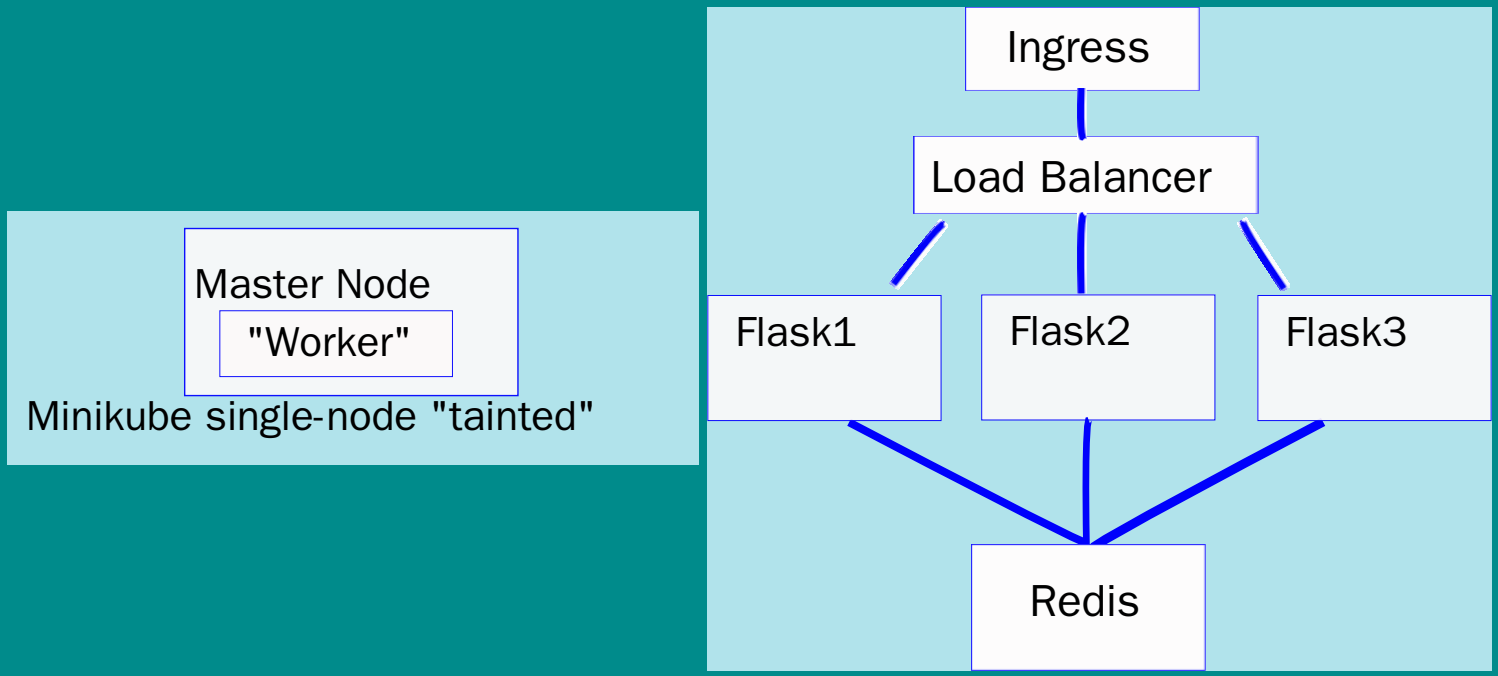
Exposing Flask Service (LoadBalancer)

```
# kubectl expose deployment flask-app --type=LoadBalancer

$ kubectl apply -f flask-service.yaml
service "flask-app" created

$ kubectl get svc
NAME         TYPE           CLUSTER-IP       EXTERNAL-IP   PORT(S)          AGE
flask-app    LoadBalancer   10.103.154.19    <pending>     5000:32201/TCP   1s
kubernetes   ClusterIP      10.96.0.1        <none>        443/TCP          5h
redis        LoadBalancer   10.101.158.201   <pending>     6379:31218/TCP   2s
```

Exposing Flask Service (LoadBalancer)

```
apiVersion: v1
kind: Service
metadata:
  labels:
    run: flask-app
  name: flask-app
spec:
  ports:
  - port: 5000
    protocol: TCP
    targetPort: 5000
  selector:
    run: flask-app
  type: LoadBalancer
```

Demo


Master Node
"Worker"
Minikube single-node "tainted"
Ingress
Load Balancer
Flask1
Flask2
Flask3
Redis

Design Pattern - Ingress

Ingress

Relation to API GW, LoadBalancer

Ingress Rules

Ingress Controller

Ingress Gateway ?

Exposing Services (Ingress)

```
$ minikube addons enable ingress
ingress was successfully enabled

$ kubectl apply -f misc/ingress-definition.yaml
ingress.extensions "ingress-definitions" created

$ sudo vi /etc/hosts
...
192.168.99.100  minikube.test flaskapp.test
```

Exposing Services (Ingress)

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: ingress-definitions
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  backend:
    serviceName: default-http-backend
    servicePort: 80
  rules:
  - host: minikube.test
    http:
      paths:
      - path: /
        backend:
          serviceName: k8sdemo
          servicePort: 8080
  - host: flaskapp.test
    http:
      paths:
      - path: /flask
        backend:
          serviceName: flask-app
          servicePort: 5000
```

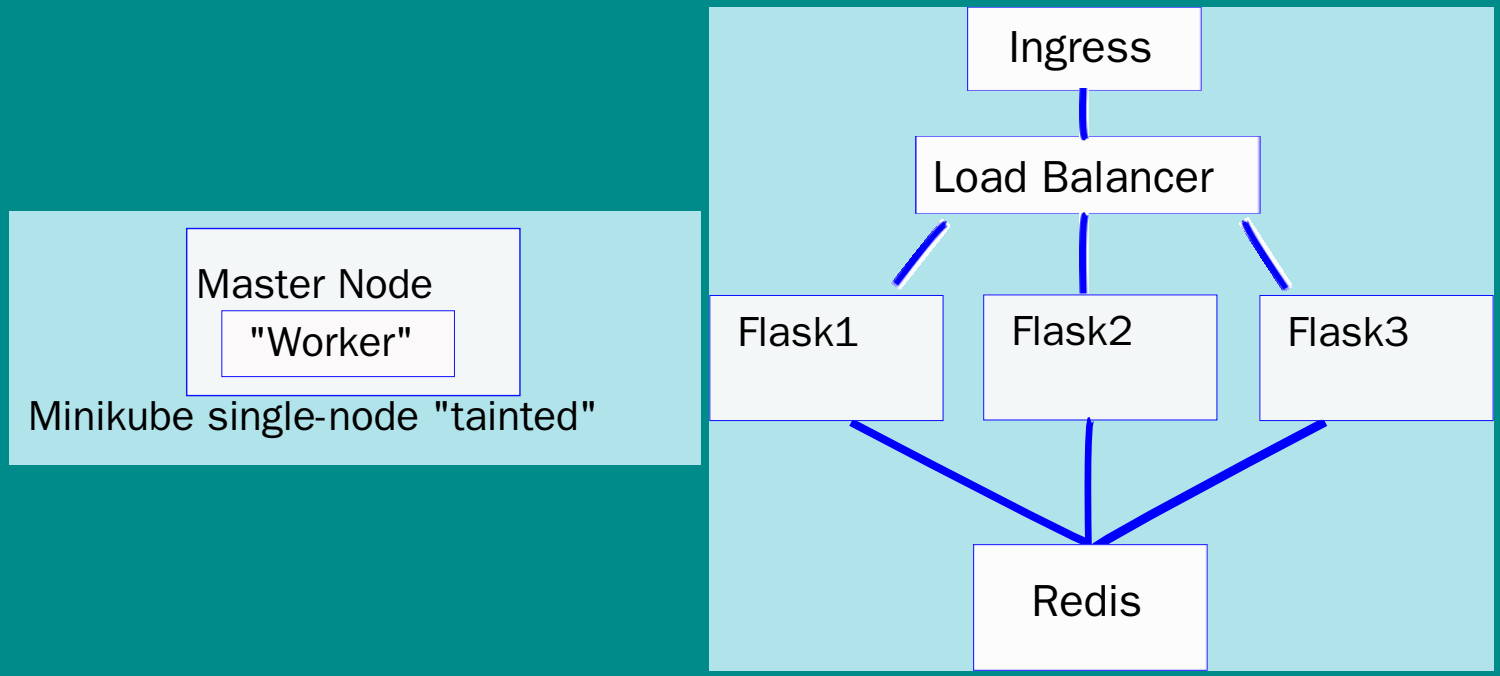
Exposing Services (Ingress)

```
$ minikube service list
|-------------|----------------------|-----------------------------|
|  NAMESPACE  |         NAME         |             URL             |
|-------------|----------------------|-----------------------------|
| default     | flask-app            | http://192.168.99.100:32201 |
| default     | k8sdemo              | http://192.168.99.100:31280 |
| default     | redis                | http://192.168.99.100:31218 |
| kube-system | kubernetes-dashboard | http://192.168.99.100:30000 |
|-------------|----------------------|-----------------------------|

$ curl http://192.168.99.100:31280

$ curl http://minikube.test/k8sdemo
```

Exposing Services (Ingress)

```
$ minikube service list
|-------------|----------------------|-----------------------------|
|  NAMESPACE  |         NAME         |             URL             |
|-------------|----------------------|-----------------------------|
| default     | flask-app            | http://192.168.99.100:32201 |
| default     | k8sdemo              | http://192.168.99.100:31280 |
| default     | redis                | http://192.168.99.100:31218 |
| kube-system | kubernetes-dashboard | http://192.168.99.100:30000 |
|-------------|----------------------|-----------------------------|

$ curl http://192.168.99.100:32201
[flask-app-8577b44db-kbwpn] Redis counter value=214

$ curl http://flaskapp.test/flask
[flask-app-8577b44db-kbwpn] Redis counter value=215
```

Demo

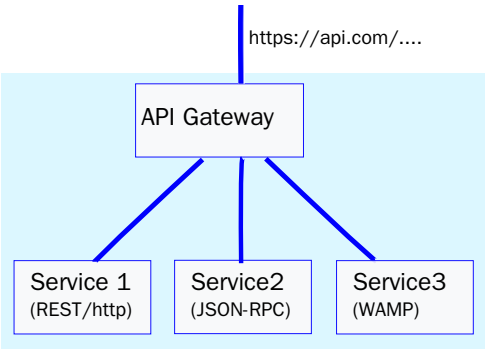

Master Node
"Worker"
Minikube single-node "tainted"
Ingress
Load Balancer
Flask1
Flask2
Flask3
Redis

????

Problem: Feel the pain

… so … **Proposition: split up components**

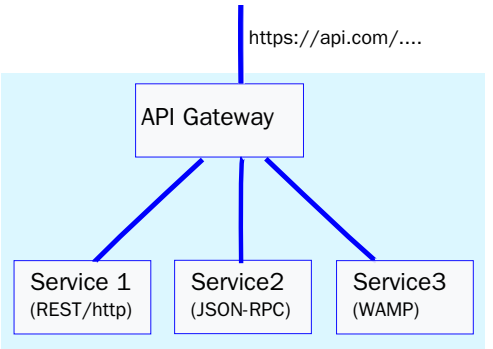
Design Pattern - API Gateway



External entrypoint exposes APIs

- Offloads common Ingress functions => reduces µ-service complexity
  - rate limiting, security, authorisation, DDOS protection
  - Protocol version translation, e.g. REST to SOAP, *-RPC ...
  - TLS decryption/encryption
- Hides internal infrastructure detail => controls access
  - service routing, load-balancing
  - Allows to refactor/scale/mock internal implementation

Design Pattern - API Gateway



External entrypoint exposes APIs

- Offloads common Ingress functions => reduces µ-service complexity
  - rate limiting, security, authorisation, DDOS protection
  - Protocol version translation, e.g. REST to SOAP, *-RPC ...
  - TLS decryption/encryption
- Hides internal infrastructure detail => controls access
  - service routing, load-balancing
  - Allows to refactor/scale/mock internal implementation

Needs to scale, be H.A.

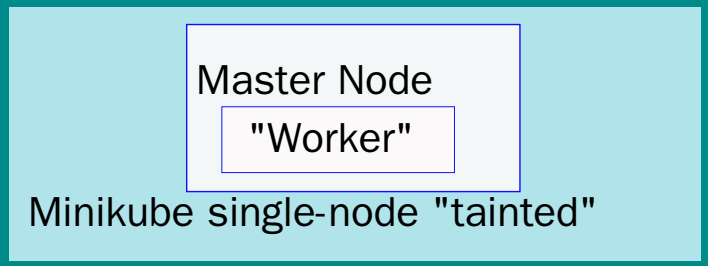
Design Pattern - API Gateway

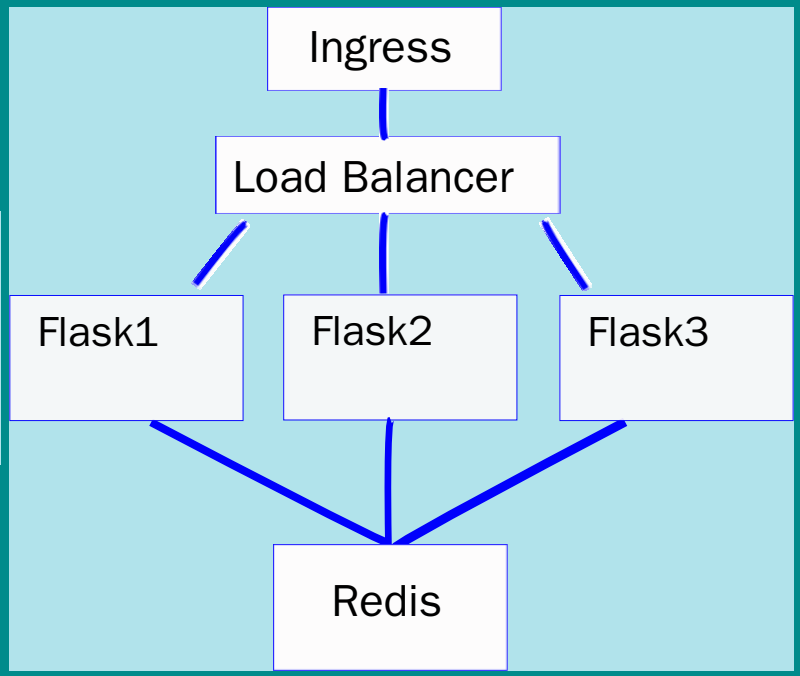
There are many API Gateways including

ha-proxy, NGInx, Traefik, AWS ELB ?!

Newer generation: Envoy-based such as Ambassador, Gloo

Demo


Master Node
"Worker"
Minikube single-node "tainted"


Ingress
Load Balancer
Flask1
Flask2
Flask3
Redis

????

Problem: Feel the pain

… so … **Proposition: split up components**

Design Pattern - Service Mesh

Abstraction above TCP/IP, secure reliable inter-service connectivity.

Platforms such as Linkerd (v2) and Istio (v1) provide offload for µ--services

Design Pattern - Service Mesh

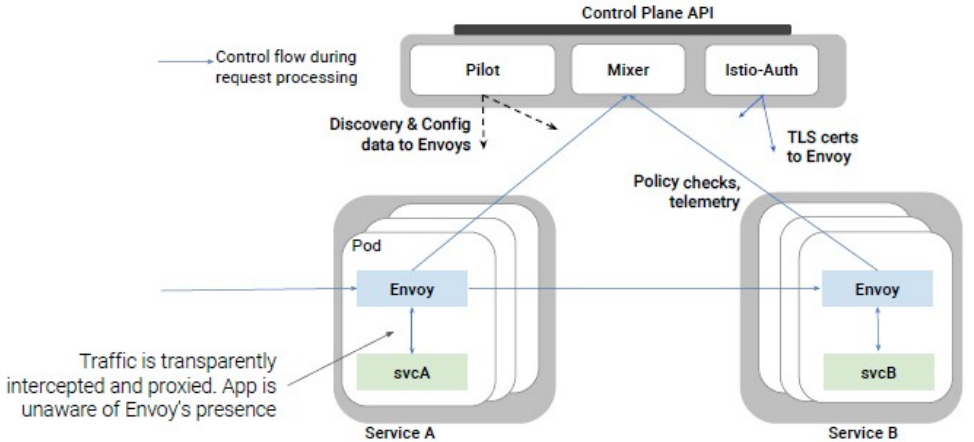
Abstraction above TCP/IP, secure reliable inter-service connectivity.

Platforms such as Linkerd (v2) and Istio (v1) provide offload for µ--services
Offloads functionality from services in a distributed way.

Design Pattern - Service Mesh

Abstraction above TCP/IP, secure reliable inter-service connectivity.

Platforms such as Linkerd (v2) and Istio (v1) provide offload for µ--services
Offloads functionality from services in a distributed way.



Design Pattern - Service Mesh - Linkerd

Abstraction above TCP/IP, secure reliable inter-service connectivity.

Platforms such as Linkerd (v2) and Istio (v1) provide offload for µ--services

Design Pattern - Service Mesh - Linkerd

Abstraction above TCP/IP, secure reliable inter-service connectivity.

Platforms such as Linkerd (v2) and Istio (v1) provide offload for µ--services

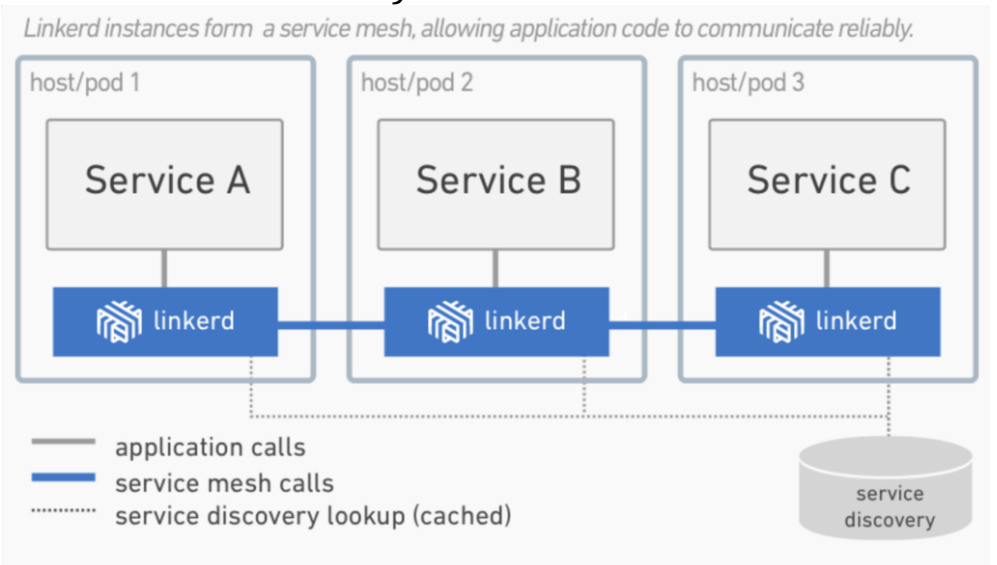
Offloads functionality from services in a distributed way.

Design Pattern - Service Mesh - Linkerd

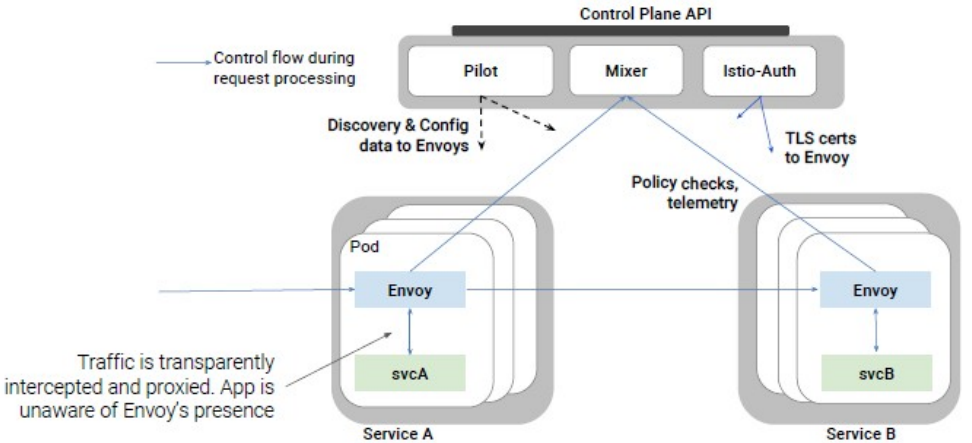
Abstraction above TCP/IP, secure reliable inter-service connectivity.

Platforms such as Linkerd (v2) and Istio (v1) provide offload for µ--services

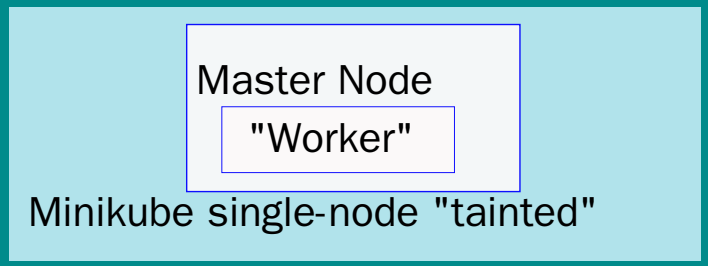
Offloads functionality from services in a distributed way.

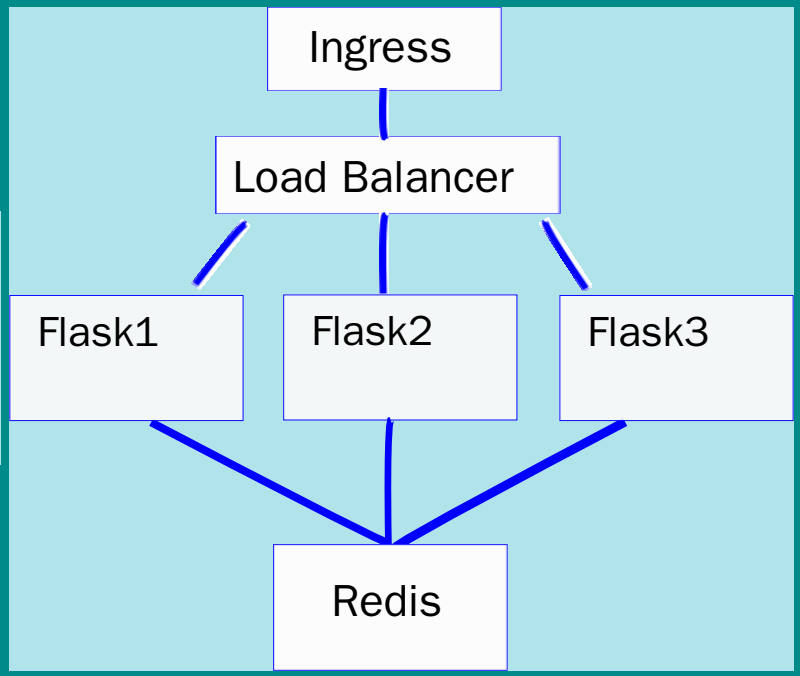


Design Pattern - Service Mesh - Istio



Demo


Master Node
"Worker"
Minikube single-node "tainted"


Ingress
Load Balancer
Flask1
Flask2
Flask3
Redis

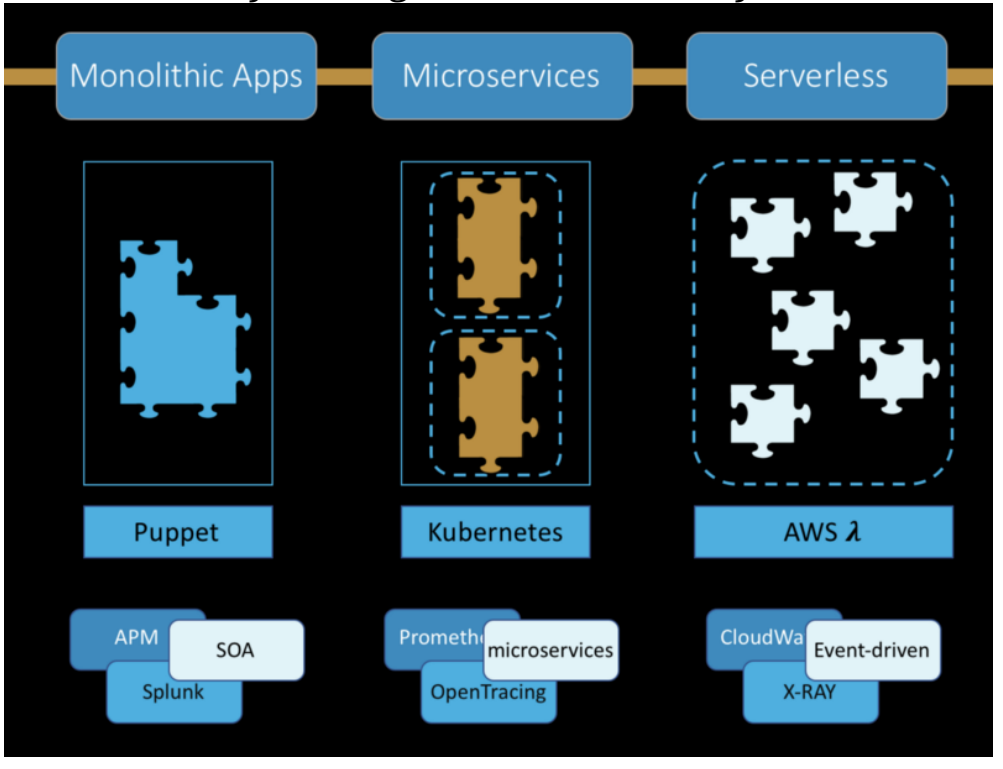
????

Problem: Feel the pain

… so … **Proposition: split up components**

Design Pattern - Hybrid Apps

Gloo allows to route between legacy apps, micro-services and serverless incrementally adding new functionality.

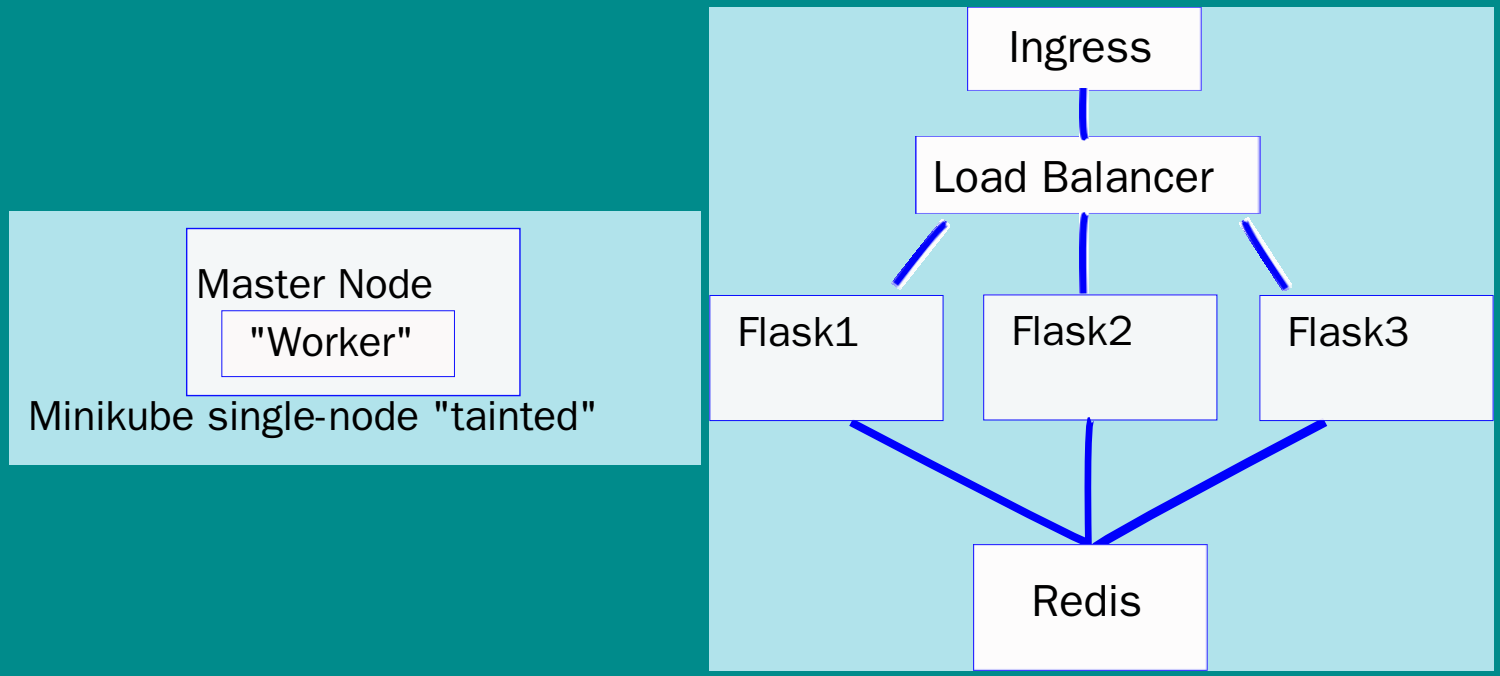


https://medium.com/solo-io/building-hybrid-apps-with-gloo-1eb96579b070

Design Pattern - Hybrid Apps

Gloo understands the infrastructure on which it is running and the APIs being used.

Demo


Master Node
"Worker"
Minikube single-node "tainted"
Ingress
Load Balancer
Flask1
Flask2
Flask3
Redis

Outline

- Monoliths to Micro-services
- Orchestration: Kubernetes
- Deployment Strategies
- Architecture Design patterns
- Summary

????

Problem: Feel the pain

… so … **Proposition: split up components**

Tools

WHAT / WHERE ????

- Tools
  - Helm (use to install tools)
  - Prometheus
  - Squash
  - Gloo
  - Istio / Service Meshes / Envoy

Summary

Micro-services offer new deployment possibilities

- with ease of deployment, scaling, upgrading

- facilitate "Best in Class" technology choices/replacements

Summary

Micro-services offer new deployment possibilities

- with ease of deployment, scaling, upgrading
- facilitate "Best in Class" technology choices/replacements

BUT moving to µ-services requires

- organizational changes and best practices !
- incremental rollout - small steps / Strangler
- hybrid approaches - old/new, cloud/on-premise, VM/container/µ-service
- offload via API Gateway and/or Service Mesh

Thank you !

From Monologue to Discussions ... ?

Questions ?

Michael Bright, @mjbright

Cloud Native Training (Docker, Kubernetes, Serverless)

Slides & source code at https://mjbright.github.io/Talks

linkedin.com/in/mjbright github.com/mjbright

Summary

Getting started with Kubernetes

- Start by learning Docker principles
- Experiment by Dockerizing some applications
- Learn about Container Orchestration
- Hands-on with Kubernetes online or Minikube(*)
- Kubernetes Visualization with KubeView
- https://github.com/mjbright/kubeview

Resources

minikube

Download	https://github.com/kubernetes/minikube/releases
Documentation	https://kubernetes.io/docs/getting-started-guides/minikube/
Hello Minikube	https://kubernetes.io/docs/tutorials/stateless-application/hello-minikube/

Resources - Articles

Martin Fowler	https://martinfowler.com/articles/microservices.html
MuleSoft, "The top 6 Microservices Patterns"	https://www.mulesoft.com/lp/whitepaper/api/top-microservices-patterns
FullStack Python	https://www.fullstackpython.com/microservices.html
Idit Levine	https://medium.com/solo-io/building-hybrid-apps-with-gloo-1eb96579b070
SSola	https://medium.com/@ssola/building-microservices-with-python-part-i-5240a8dcc2fb
Deployment	http://container-solutions.com/kubernetes-deployment-strategies/

TO ADD: 12-factor apps Gloo - Christian Posta

Resources - Books

Publisher		Title, Author
O'Reilly		"Building Microservices", Sam Newman, July 2015
PacktPub		"Python Microservices Development", Tarek Ziade, July 2017

kNative - O'Reilly

Istio - Manning

Istio - O'Reilly

Testdriven.io