Microservices

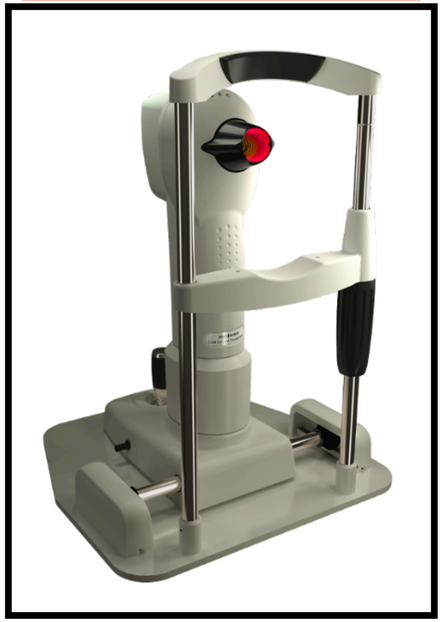
Patterns, implementation and deployment

Jake Brown - jake@eyespacelenses.com

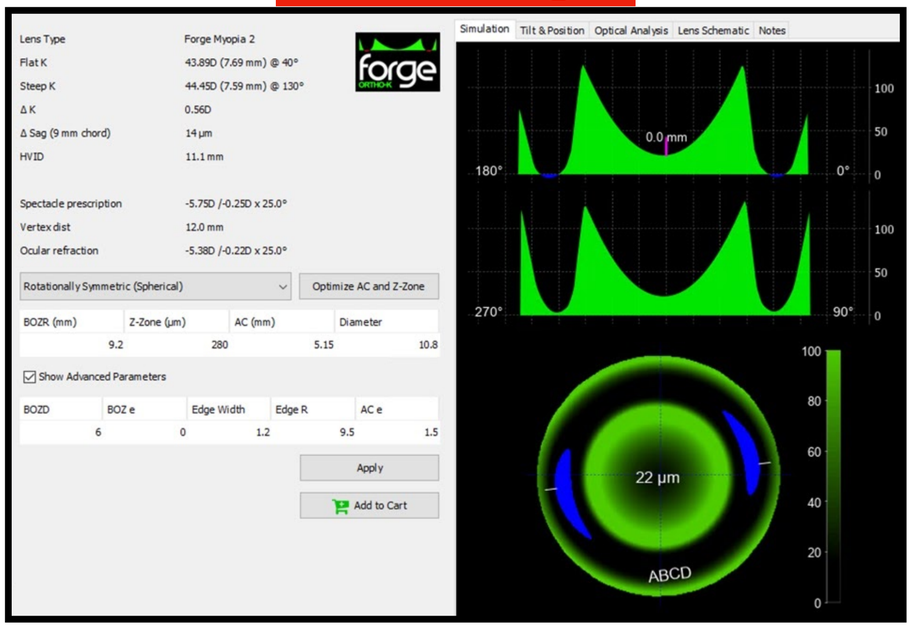
Jake Brown

- University of Adelaide class of 2012 (High Performance Computational Physics Hons)
- Self taught in Software Architecture
- Spent more time learning FORTRAN than looking at how to build and deploy software at scale
- Co-founded EyeSpace™ in 2012

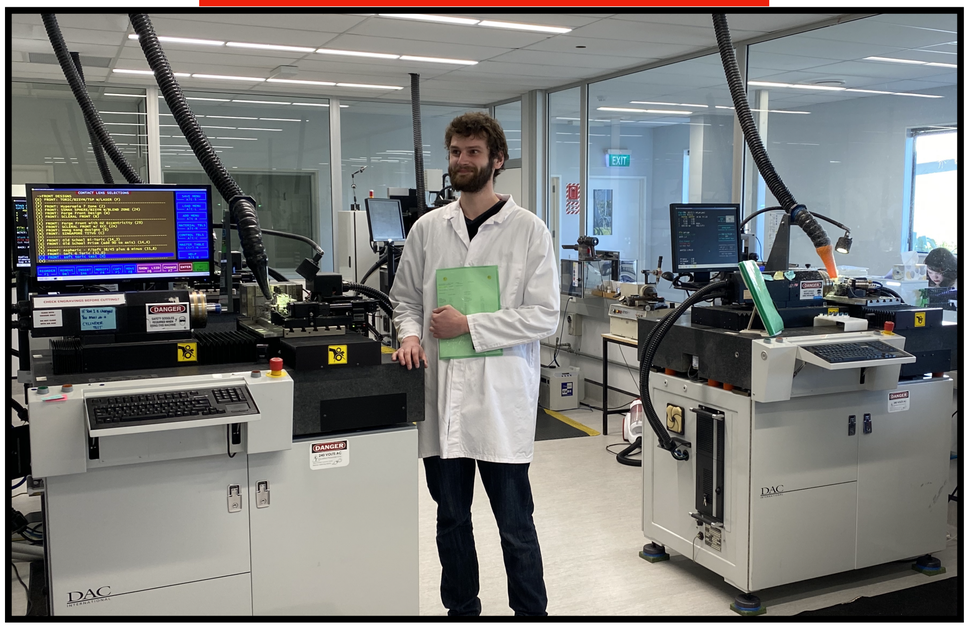
1: Capture



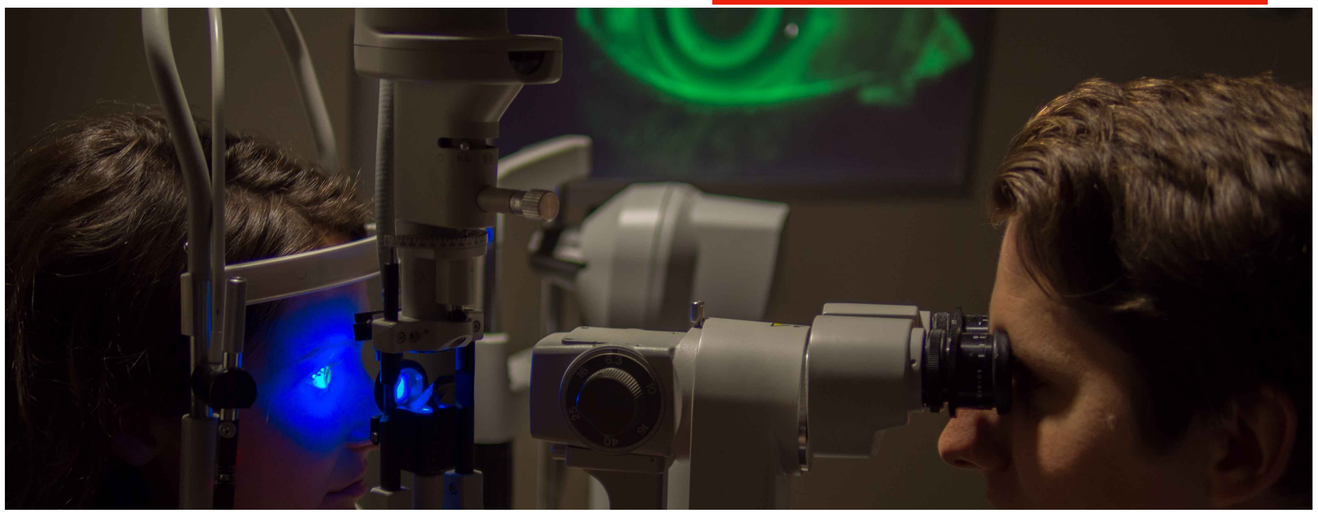
2: Design

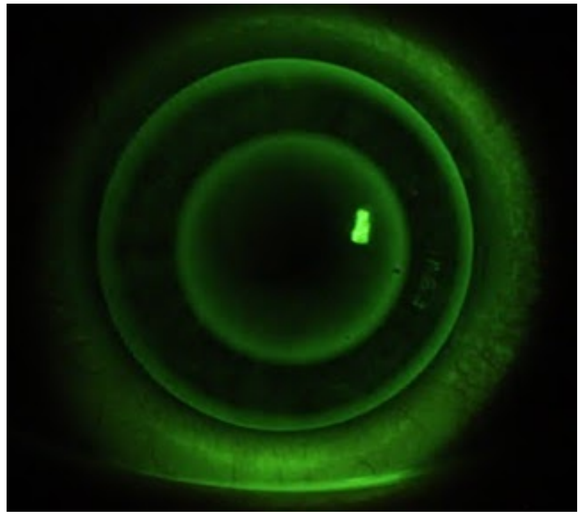


3: Manufacture

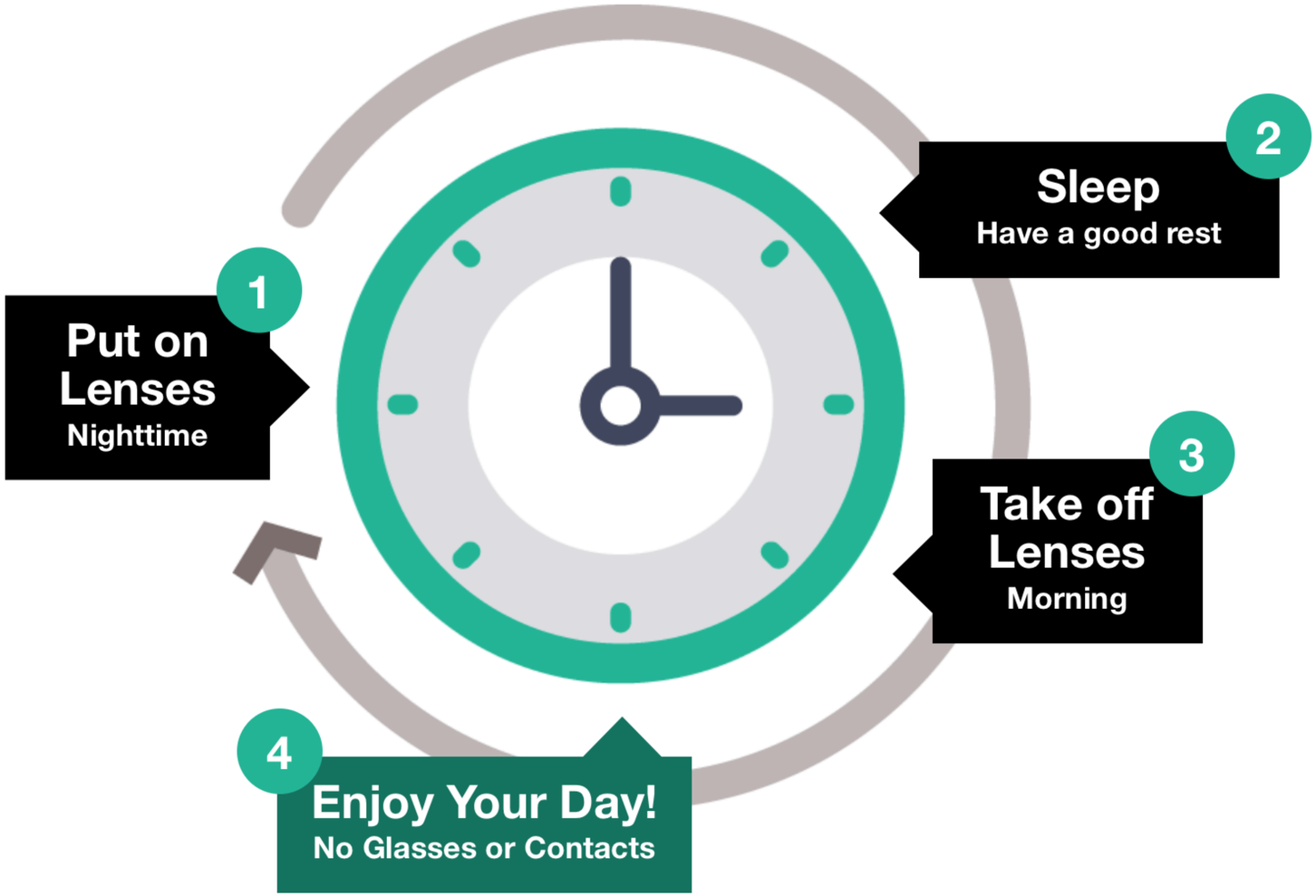


4: Assessment

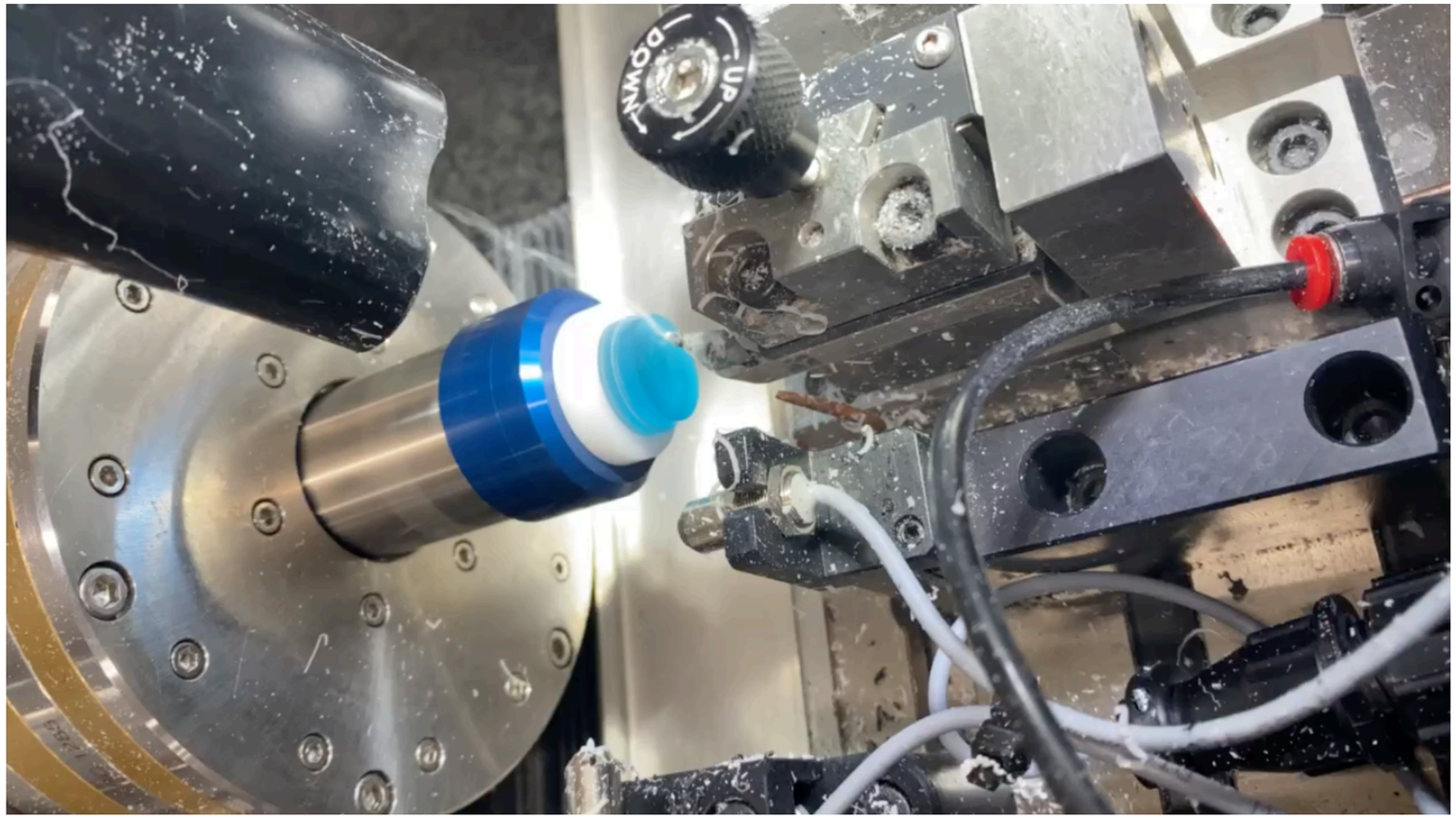




5: Long-term wear



Video: lathe



https://jakebrown.io/videos/lathe-oscillating.mp4

Which clinics supply our lenses?

EyeSpace™ BY INNOVATUS TECHNOLOGY

HIPAA (USA)

The Health Insurance Portability and Accountability Act of 1996

- Applies anywhere you are using Protected Health Information (PHI)
- Data must be encrypted in transit and at rest
- Event and access logging must be retained for 6 years
- MUST have a signed Business Associates Agreement (BAA) with the host

Our team

- I **don’t** have experience building enterprise grade software with hundreds of developers and multiple teams - we have one small team of three developers
- New technologies and patterns, like microservices, are making it easier for small teams to enter markets typically reserved for large enterprise

Microservices

Microservices vs SOA

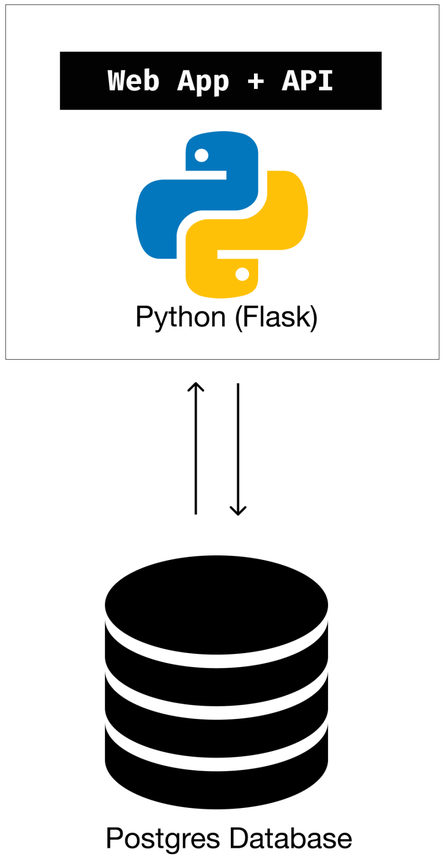
- Service-oriented architectures (SOA) are traditionally applied to expose and **reuse applications across an enterprise** (enterprise scope)
- Microservices are used to **build applications** (application scope)
- I'm going to be talking about software architecture using microservices in a small team and at the ***application* scale**

> "The first "generation" of service-oriented architectures (SOA) defined daunting and nebulous requirements for services (e.g., discoverability and service contracts), and this hindered the adoption of the SOA model. Microservices are the second iteration on the concept of SOA… **The aim is to strip away unnecessary levels of complexity in order to focus on the programming of simple services that effectively implement a single functionality**."

Dragoni, Nicola & Giallorenzo et al (2017). Microservices: yesterday, today, and tomorrow.

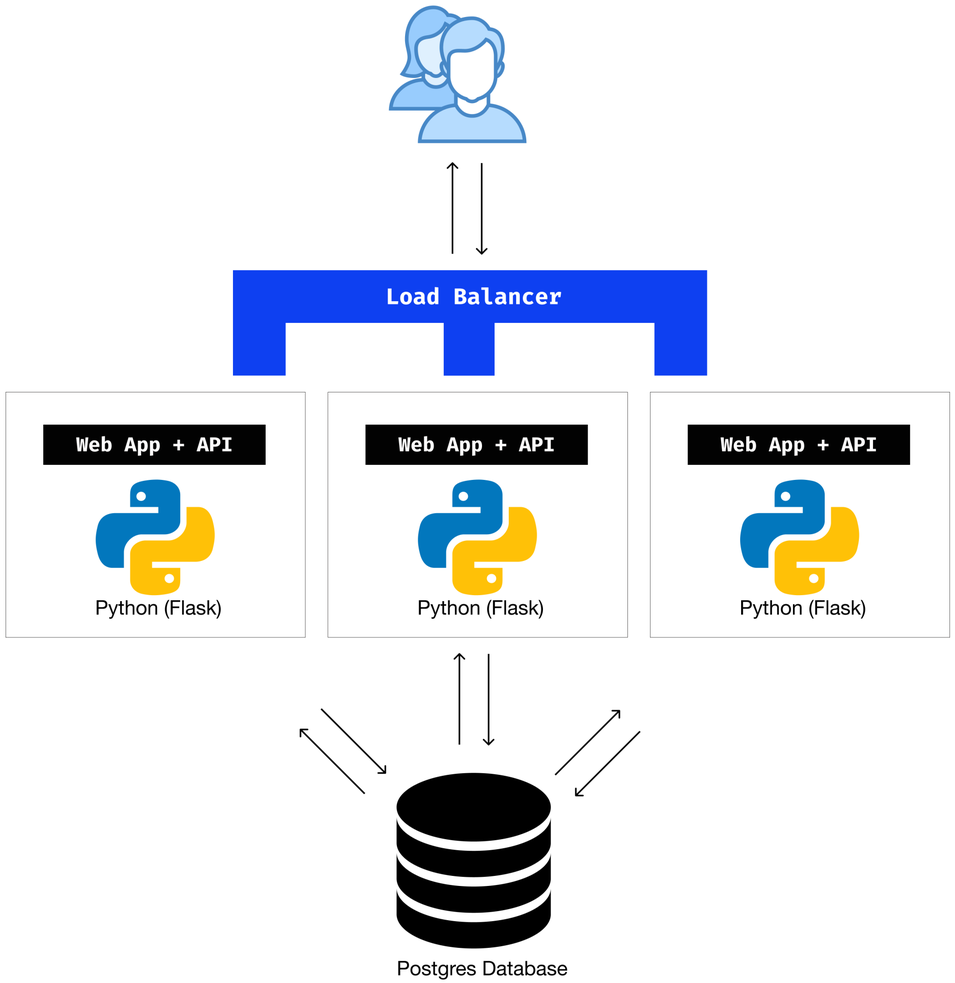
Stateful Monolith

- Monolith => one large application
- Stateful => running application can store information (eg. user sessions) in memory, filesystem etc
- Being stateful means that we can only run one copy of the app
- Database may be installed on the same server
- Can only scaled up (add CPU+memory)
- Very simple and (historically) popular architecture (eg. Wordpress)



Stateless Monolith

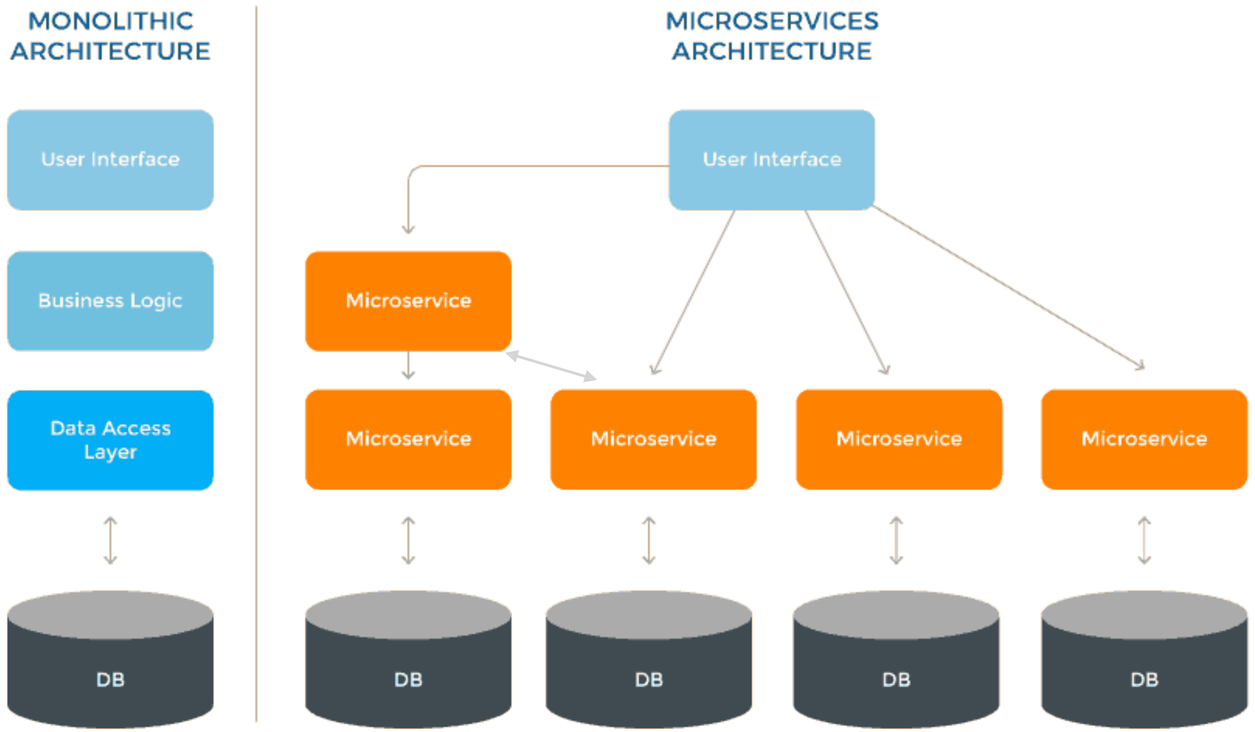
- Stateless => each request is independent
- Any state is stored in database or client-side
- Stateless architecture enables:
  - Fault tolerance
  - Rolling updates
  - Scaling **out** (add servers)



https://cloud.google.com/solutions/twelve-factor-app-development-on-gcp

Microservices

- We can decouple our monolith into separate (ideally) single-purpose services
- Each microservice (usually) has its own database
- Microservices can call each other, or the UI can interact with them directly
- Each can be built, run, deployed, and scaled separately



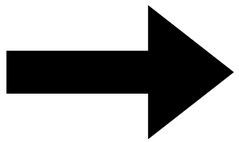
https://www.plutora.com/blog/understanding-microservices

EyeSpace

Microservices approach

Web App + API

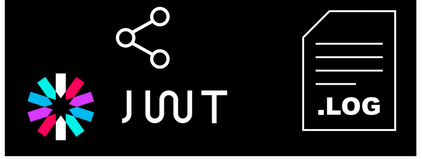
Python (Flask)



Front-end client

PHI Service

Auth Service



Order Service

Notification Service

Desktop Software API

Manufacturing Service

Billing Service

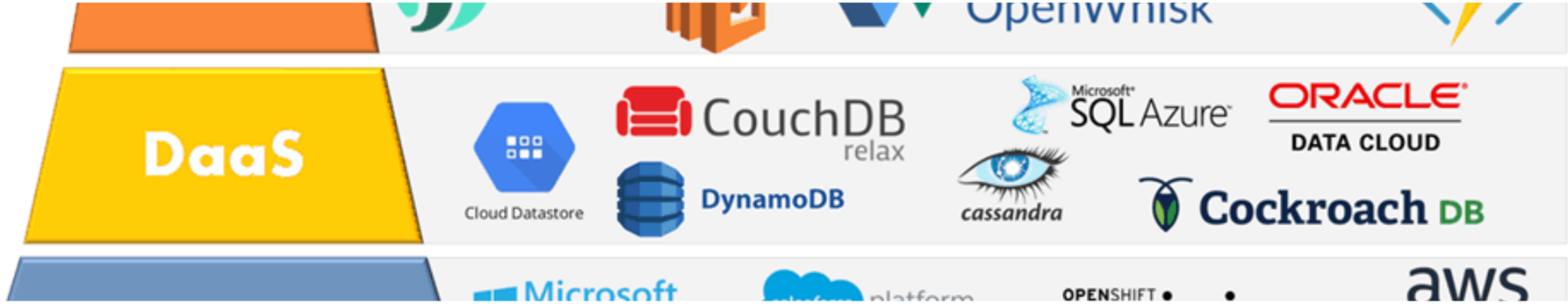
Microservices

Why?

- **Security and compliance**: eg. Protected Health Information
- **Performance**: Different programming languages or different hardware
- **Availability:** Uptime requirements might be different
- **Legacy software:** Expose functionality without rewriting legacy code
- **Different teams:** Clear boundaries between code-bases

Databases

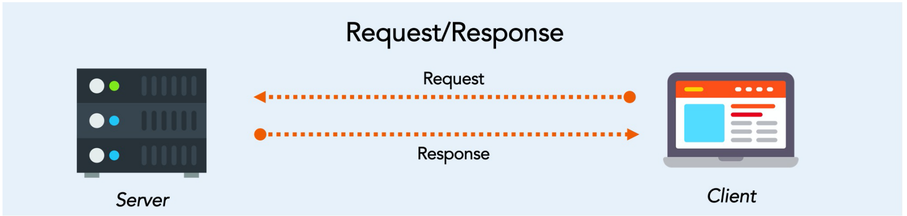
- Each microservice usually has it’s own database and data model
- Could be SQL or new breed of cloud-native databases
- Data *can* be denormalised: duplicate data can stored across different tables or different databases
- Sometimes stored according to access patterns or security rules



Microservice communication

Synchronous request/response pattern

- HTTP provides a well defined synchronous request/ response protocol
- Request:
  - **method** (GET, POST, PUT, PATCH, DELETE…)
  - **path**
  - optional: **headers** and **body**
- Response:
  - **status code**
  - optional: **headers** and **body**



Request

```
/users/1 [GET]
```

Response

```
200
Content-Type: application/json

{
    "user": {
        "id": 1,
        "name": "Jake",
        "country": "Australia"
    }
}
```

HTTP status codes

- Status code can indicate retry-ability
- Which codes should we retry?

	Request	Response
1	`/users/1 [GET]`	`503`
2	`/users/1 [GET]`	`401`

3	`/users/1 [GET]`
	`Authorization:Bearer eyJhbGc…`
		`200`
		`Content-Type: application/json`
		
		`{`
		`  "user": {`
		`  "id": 1,`
		`  "name": "Jake",`
		`  "country": "Australia"`
		`  }`
		`}`

HTTP STATUS CODES

2xx Success

- 200 Success / OK

3xx Redirection

- 301 Permanent Redirect
- 302 Temporary Redirect
- 304 Not Modified

4xx Client Error

- 401 Unauthorized Error
- 403 Forbidden
- 404 Not Found
- 405 Method Not Allowed

5xx Server Error

- 501 Not Implemented
- 502 Bad Gateway
- 503 Service Unavailable
- 504 Gateway Timeout

INFIDIGIT

https://www.infidigit.com/blog/http-status-codes/

REST

REpresentational State Transfer

- Often “HTTP API” and “REST API” are used interchangeably
- REST is more formal, introduces constraints:
  - Strict relationship between a path and a resource
  - Hypermedia as the Engine of Application State (HATEOAS)

Request

```
/users/1 [GET]
```

Response

```
HTTP/1.1 200 OK

{
    "user": {
        "id": 1,
        "name": "Jake",
        "country": "Australia",
        "links": {
            "posts": "/users/1/posts"
        }
    }
}
```

HATEOAS

Safety and Idempotence

- **Safety**: does the request modify the application state
- **Idempotence**: if the request is performed more than once, does this change the application state?
- Implementing REST gives you these guarantees

HTTP Method	Idempotence	Safety
GET	YES	Yes
HEAD	YES	Yes
PUT	YES	No
DELETE	YES	No
POST	No	No
PATCH	No	No

- What HTTP Method would I use if I want to *add* $5 to my bank account balance?

	Request	Response

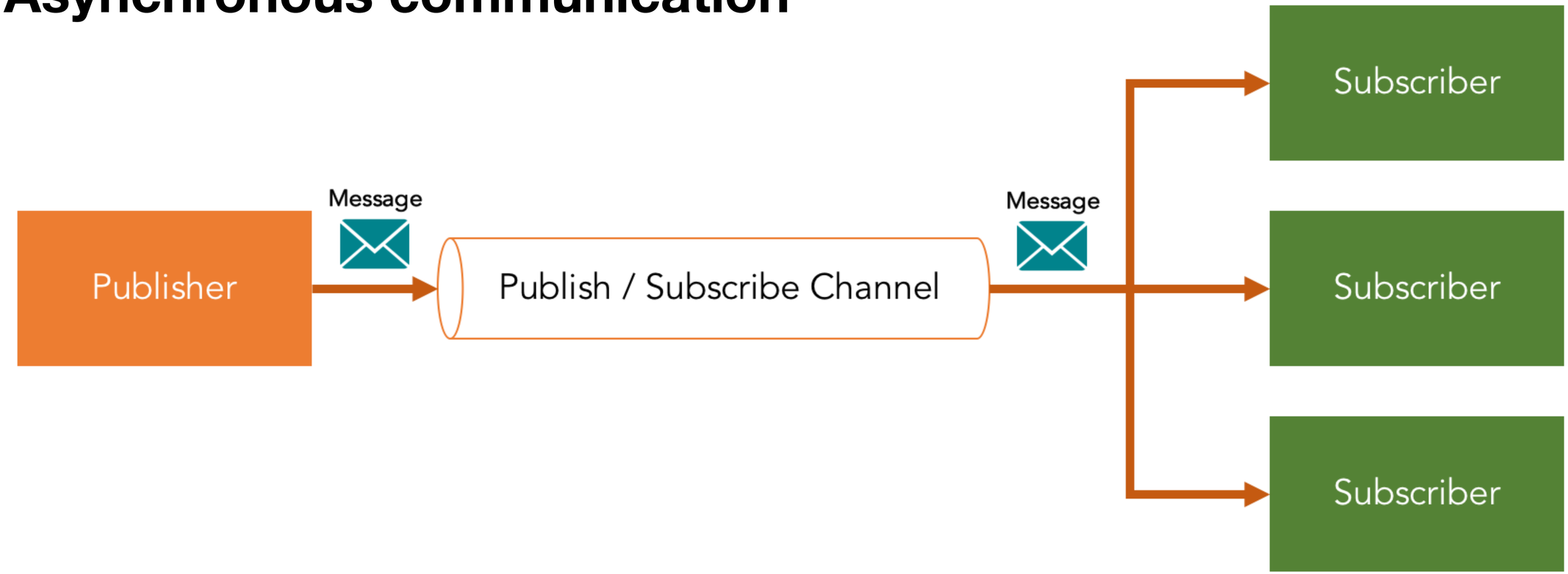
GraphQL	`query GetUserAndPosts {`
	`  getUser(id: 1) {`
	`  id`
	`  name`
	`  posts(first: 10) {`
	`  id`
	`  title`
	`  }`
	`  }`
	`}`
		`{`
		`  "user": {`
		`  "id": 1,`
		`  "name": "Jake",`
		`  “posts": [`
		`  {"id": 1,"title": "My first Post”},`
		`  {"id": 2, "title": "My second Post”}`
		`  ]`
		`  }`
		`}`

REST request 1	`/users/1 [GET]`	`HTTP/1.1 200 OK`
		`{`
		`  "user": {`
		`  "id": 1,`
		`  "name": “Jake”,`
		`  “country": "Australia",`
		`  "links": {`
		`  "posts": "/users/1/posts"`
		`  }`
		`  }`
		`}`

REST request 2	`/users/1/posts [GET]`	`HTTP/1.1 200 OK`
		`{`
		`  “posts": [{`
		`  "id": 1,`
		`  "title": "My first Post",`
		`  "date": "1/12/2019",`
		`  "content": "Lorem ipsum dolor sit amet…"`
		`  }, {`
		`  "id": 2,`
		`  "title": "My second Post",`
		`  "date": "1/1/2020",`
		`  "content": "Duis aute irure dolor…"`
		`  }, …]`
		`}`

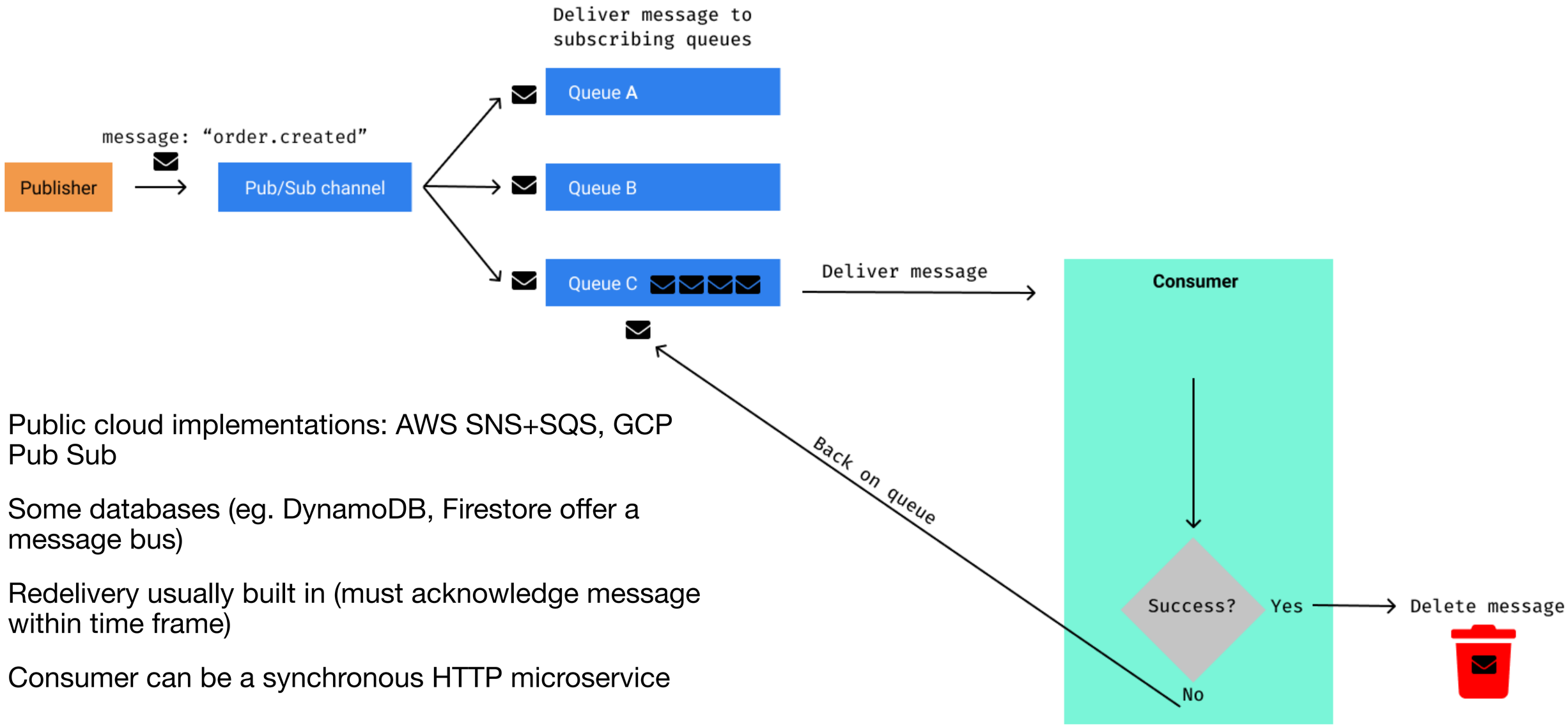
Patterns - Pub/Sub

Asynchronous communication



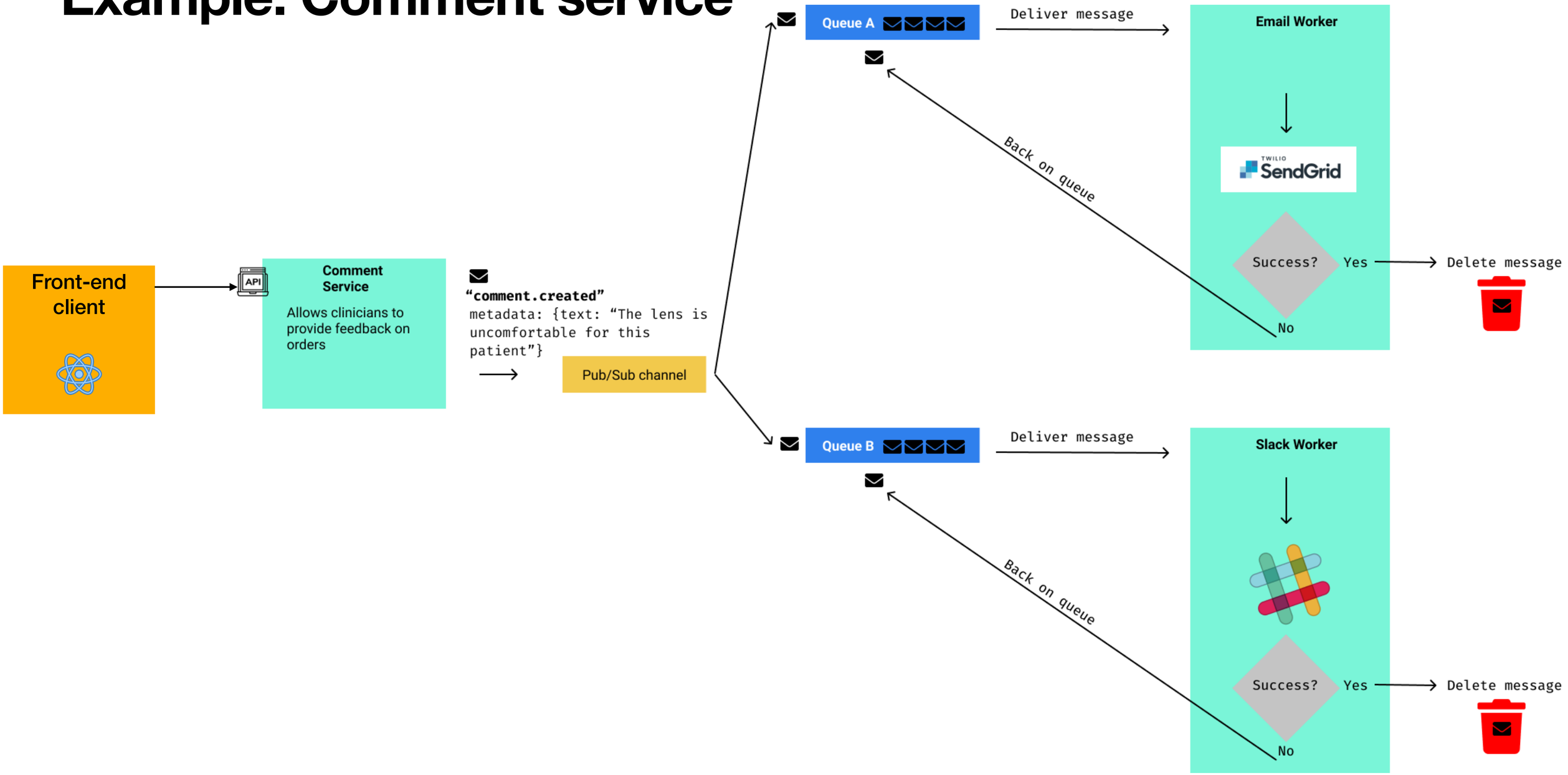
https://realtimeapi.io/hub/publishsubscribe-pattern/

Pub/Sub implementation



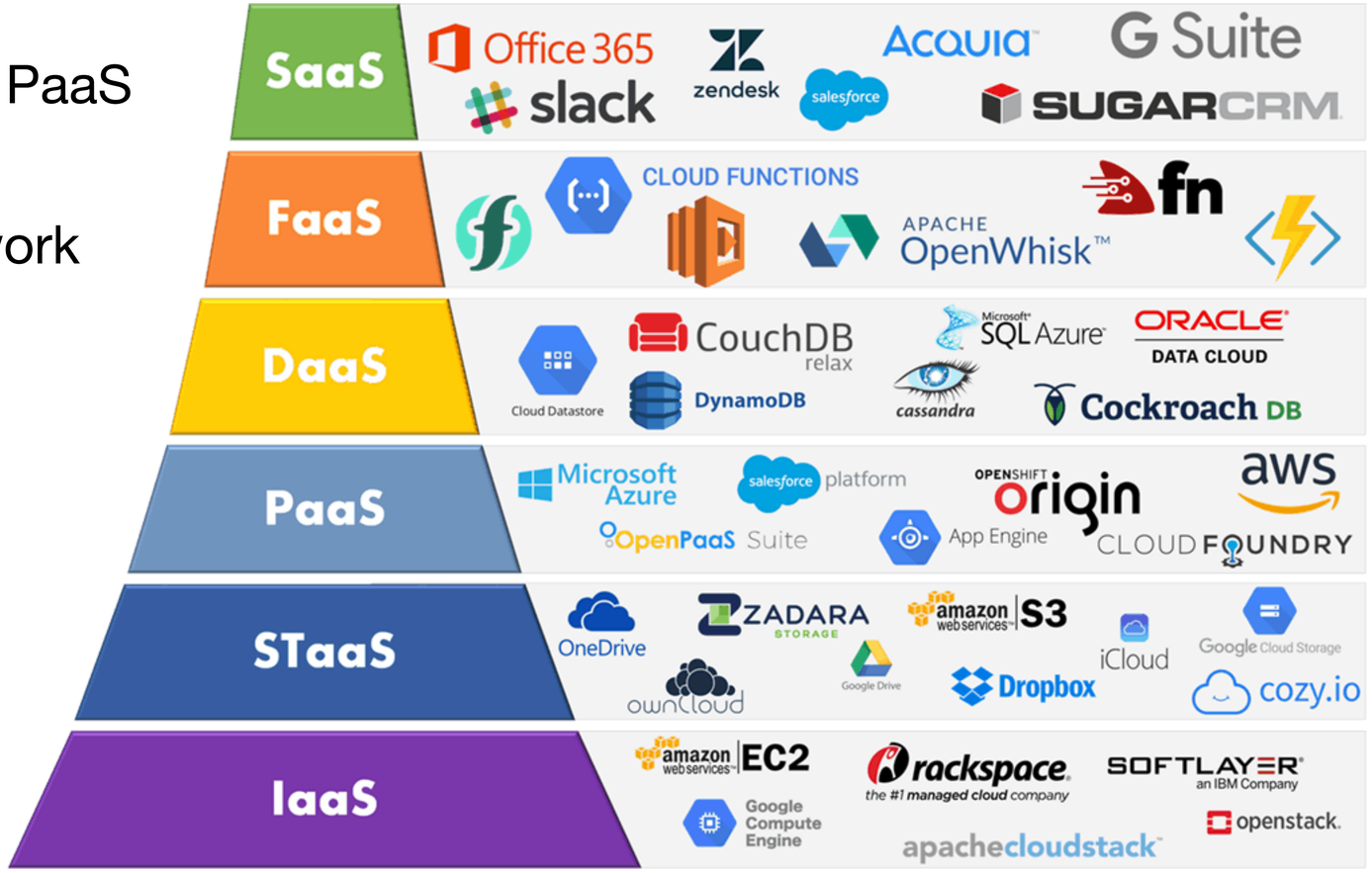
- Public cloud implementations: AWS SNS+SQS, GCP Pub Sub
- Some databases (eg. DynamoDB, Firestore offer a message bus)
- Redelivery usually built in (must acknowledge message within time frame)
- Consumer can be a synchronous HTTP microservice

Example: Comment service



Deployment

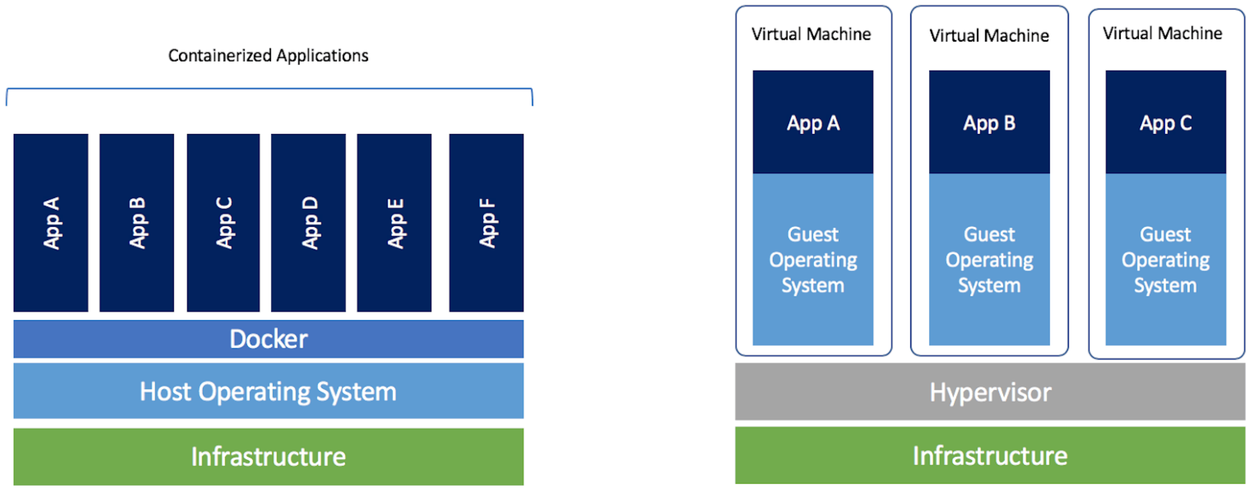
- Nearly all applications will use STaaS and/or DaaS for persistence
- For microservices, we could use FaaS, PaaS or IaaS
- CaaS: more portable than PaaS, less work than IaaS



https://imelgrat.me/cloud/cloud-services-models-help-business/

Containerisation

- Bundles an app and any dependent packages in one easy to run package - an “image”
- You can think of it like like lightweight virtualisation WARNING: That analogy doesn’t adequately cover the security implications
- It’s sharing the underlying host kernel



https://www.docker.com/blog/containers-replacing-virtual-machines/

Docker example

main.py

```
from flask import Flask
app = Flask(__name__)

@app.route("/healthz")
def healthz():
    return "OK", 200

@app.route('/hello')
def hello_endpoint():
    return "{'msg':'Hello, World'}" , 200,
{'Content-Type': 'application/json'}
```

Dockerfile

```
FROM python:3.7.9-stretch

# Copy over the application code
RUN mkdir /app
COPY main.py /app/
WORKDIR /app
RUN pip install flask

# Configure the application
ENV FLASK_APP main.py
CMD ["flask", "run", "--port", "5000",
"--host", "0.0.0.0"]
```

```
$ docker build . -t hello:v1
> Successfully built hello:v1

$ docker run -p 8080:5000 hello:v1
> Running on http://0.0.0.0:5000

$ curl http://127.0.0.1:8080/hello
 > {'msg':'Hello, World’}
```

Images and Containers

- You *build* the Dockerfile to create an *image*:

```
docker build . -t hello:v1
> Successfully built hello:v1
```

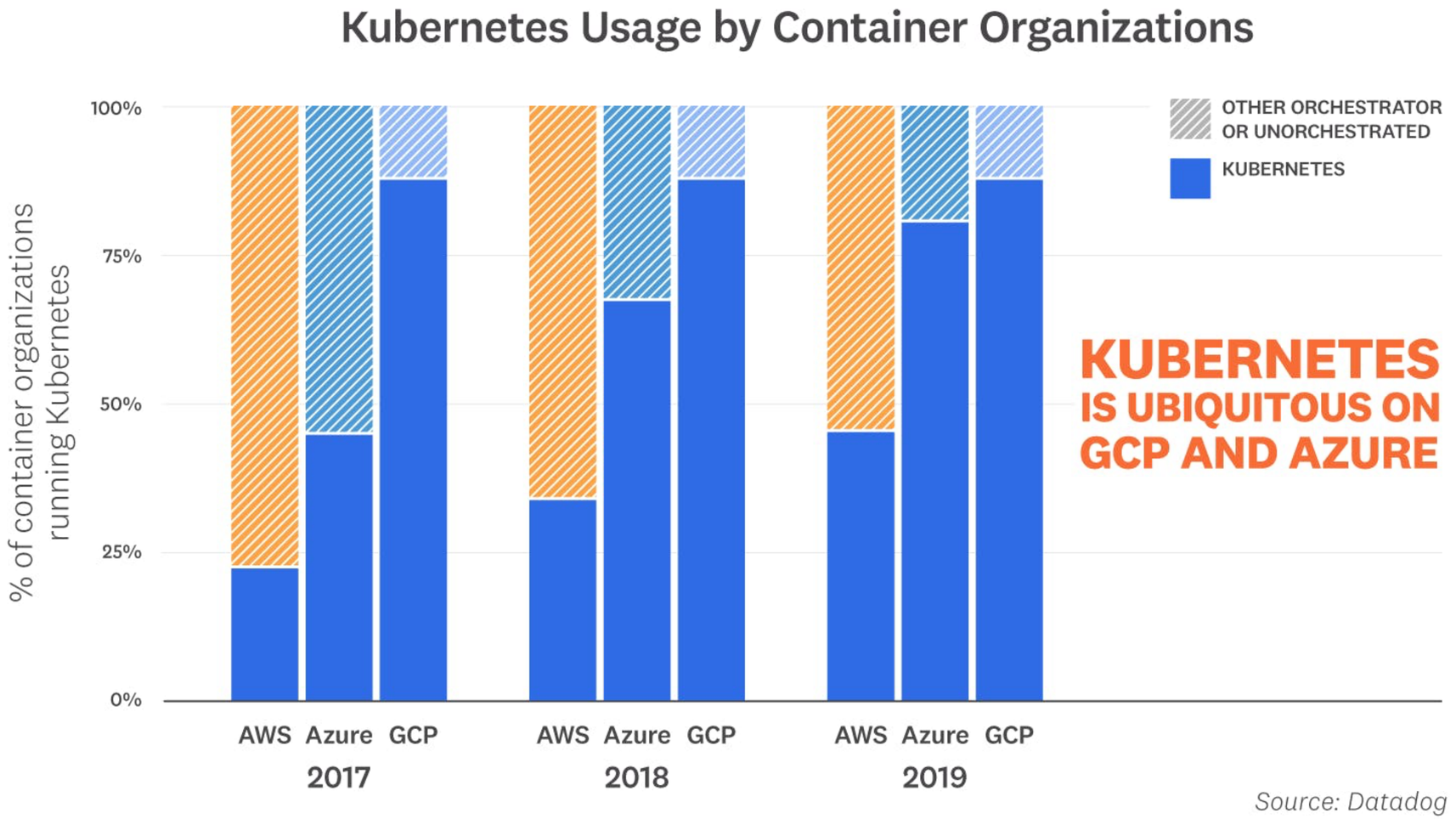
- You *run* an *image* to create a *container*:

```
docker run -p 8080:5000 hello:v1
> Running on http://0.0.0.0:5000
```

Container Orchestration

- We need a way of running *many* containers, networking them, and restarting them when they fail
- Either on a single machine or across a cluster
- A container **orchestrator** is like an Operating System for containers and handles all of this
- **Imperative**: Tell the orchestrator what to do
  - eg. “Start a *hello-world* container for me”
  - More interactive
- **Declarative**: Tell the orchestrator what you want
  - eg. “I want three copies *hello-world* container running at all times”
  - Allows infrastructure as code and is more reproducible

Kubernetes (K8)

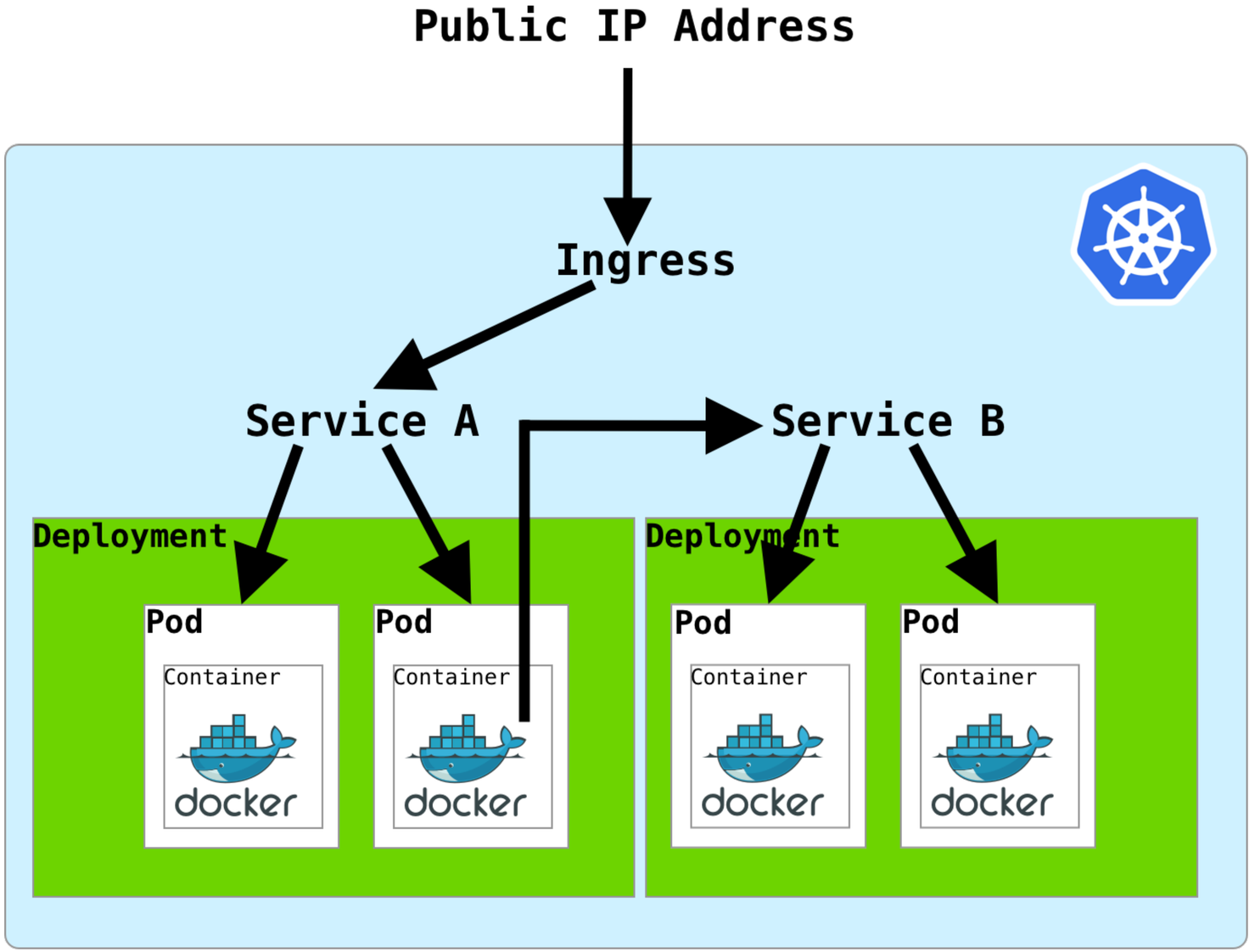


Alternatives: Docker Swarm, Apache Mesos, AWS ECS…

https://www.datadoghq.com/container-report/

Kubernetes concepts

- **Pod** = container (simplified)
- **Deployment** adds scaling and automatic restarts
- **Service** allows pods *inside* the cluster to communicate with each other
- **Ingress** punches a hole into the cluster and exposes the service



K8 Deployment

- Take our docker image from before (hello:v1)
- Wrap it in a deployment to give it scaling, health checks, restarts etc
- Define livenessProbe

deployment.yaml

```yaml
kind: Deployment
metadata:
    name: hello-deployment
spec:
    replicas: 2
    template: # pod template
            labels:
             app: hello-app
            containers:
              - name: hello-container
                image: us.gcr.io/demo-project/hello:v1
                ports:
                  - containerPort: 5000
                livenessProbe:
                     httpGet:
                          path: /healthz
                          port: 5000
                     initialDelaySeconds: 5
                     periodSeconds: 5
```

K8 Service and Ingress

service.yaml

```
kind: Service
metadata:
    name: hello-svc
spec:
    type: NodePort
    ports:
      - port: 5000
        name: http
        targetPort: 5000
    selector:
      app: hello-app
```

- Now we can do this inside the cluster:

```
$ curl http://hello-svc:5000/hello
 > {'msg':'Hello, World’}
```

K8 Service and Ingress

ingress.yaml

```
kind: Ingress
metadata:
  name: ingress-example
spec:
  rules:
  - http:
      paths:
      - path: /*
        backend:
          serviceName: hello-svc
          servicePort: 5000
```

- Now we can do this outside the cluster:

```
$ curl http://cluster-ip:5000/hello
 > {'msg':'Hello, World’}
```

Now deploy it!

```
$ docker build . -t hello:v1 # Build the image
$ docker push demo-project/hello:v1 # Push to image registry
$ kubectl apply -f deployment.yaml service.yaml ingress.yaml
```

- If you’re using Kubernetes as a service, this is the first and final step
- Not the case if you want to install and maintain your own Kubernetes cluster…

Microservices challenges

- Can’t run entire application locally… but could you do that anyway?
- If services are too tightly coupled, it will be difficult to develop them independently
- If you’re really struggling with this you could be (accidentally) building a distributed monolith
- Logging and tracing becomes difficult in a distributed system
- Security - more network complexity = more possible attack vectors
- Important to automate deployment (DevOps, GitOps, CI/CD)

Cloud native summary

- Cloud Computing v1 changed *where* we *host* software
- **Cloud Native** represents a rethink of the way we *build* and *deploy* software
- Not just taking our old methods and moving them onto the cloud - it's a new way of designing and building software
- You don't have to use K8 or even containers: other PaaS options may be simpler if vendor lock-in is not a big deal
- Hopefully we'll see more standardisation across public cloud providers in the coming years
- We're already seeing some PaaS built on top of K8 eg. Knative and Google Cloud Run

We are hiring!

Contact **jake@eyespacelenses.com**

EyeSpace™

bespoke EyeSpace™

forge ORTHO-K

Scleral EyeSpace™

eyespacelenses.com